AFIT/GCS/ENG/92D-23

DESIGN RECOVERY FOR SOFTWARE LIBRARY POPULATION

THESIS

Chester A. Wright, Jr.
Captain, USAF

AFIT/GCS/ENG/92D-23

AFIT/GCS/ENG/92D-23

DESIGN RECOVERY FOR SOFTWARE LIBRARY POPULATION

THESIS

Presented to the Faculty of the School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirements for the Degree of

Master of Science (Computer Engineering)

Chester A. Wright, Jr., A.A.S., B.S.

Captain, USAF

December, 1992

Approved for public release; distribution unlimited

Preface

The purpose of the study was to investigate the possibility of recovering the design of existing software to populate a reuse library. The immediate need is to populate the Automatic Programming Technologies for Avionics Systems (APTAS) library, but the approach used should be valid for general library population.

There were many people that helped me tremendously throughout this project. I would like to express my appreciation to my sponsor, John Werthmann from Wright Laboratories/AART, for allowing me the opportunity to learn and discover while applying school studies to a real-world problem. I am most grateful to my advisor, Captain James Cardow for his patience and perseverance in his effort to lead and guide me in the right direction. Finally, many thanks and my love to my wife Tami for her concern, pressure, and understanding during the many early mornings of study.

Chester A. Wright, Jr.

Table of Contents

Page

List of Figures

AFIT/GCS/ENG/92D-23

Abstract

The thesis research investigated design recovery as a means of populating a reuse library. The targeted library was part of the Automatic Programming Technologies for Avionics Systems (APTAS). APTAS uses a knowledge base of forms, to present questions to a user, and rules, to select the forms to present and choose existing library modules to use in composing a new system. The approach applied the reengineering model developed by Eric Byrne to accomplish planning for the project, expanded the renovation phase of this model to cover the actual design recovery, and applied the expanded model to populating the library.

Using the model in the project showed that design recovery is feasible in populating the library. However, if the recovered design could not be used directly, it could be used as a guide in developing new components. Additionally, certain modules make better candidates than others. Ideal candidates are self-contained in that they receive a value, perform a computation, and return a value. Once the module starts performing too many operations, expertise is required in the module behavior in order to separate the component for reuse.

DESIGN RECOVERY FOR SOFTWARE LIBRARY POPULATION

I. Introduction

1.1 Background

Developing any system requires defining what the system is to accomplish, identifying resources required to build the system, and putting the resources together to produce the desired product. Once the system is in existence, it is used for the intended purpose, except during periods when the system requires maintenance. The waterfall model, shown in Figure 1.1, represents all of these processes as phases in the lifecycle of a software system. The analysis phase defines the user's requirements. Actions in this phase provide a detailed description of the problem that needs to be solved; documents information flow and structure in the current environment; and describes hardware, software and human interfaces as they will exist (16). The product of this phase is a document, called the program specification, listing all of the requirements the system is to accomplish. Using this specification as input, the design phase translates each requirement into a software representation (16). Additionally, required system resources are identified. The output of this phase is the design document outlining the necessary software modules and their functionality. At the code phase, programmers take the design document and convert each module into a form that can be executed on the computer system. During test, system behavior is compared with the requirements to ensure there is a correct correspondence. Once the system leaves the test phase, it passes to the customer and is used and maintained as necessary.

Developing software using the waterfall model has created problems. General Randolph, as quoted in (2), summed up the nature of the problem with software development when he said

"We've a perfect record...: we've never made one on time yet".

In his studies of software development problems, Brooks (6) refers to work by Charles Portman, manager of ICL's Software Division. Portman initially found his programming teams were taking twice as long as expected. When this slippage pattern appeared, he asked his teams to keep careful

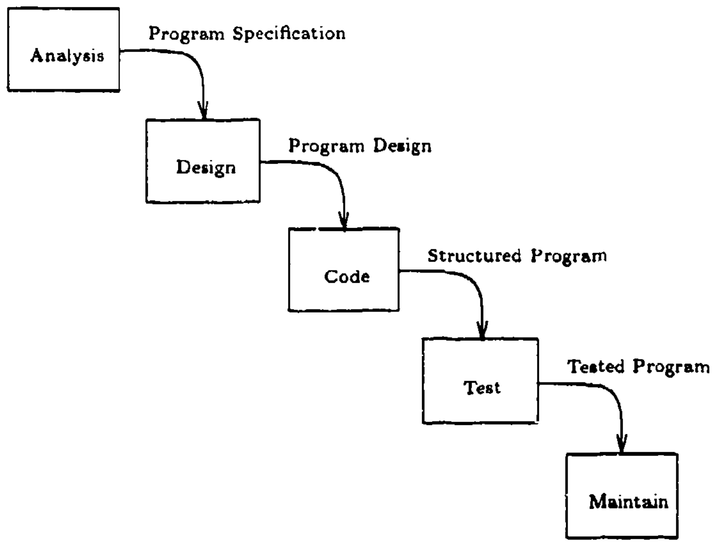


Figure 1.1. Classical Software Lifecycle Model

logs of time usage and found the teams were only able to utilize 50% of the work week as actual programming and debugging time. Machine downtime, meetings, paperwork, company business, and higher-priority, short, unrelated jobs were among the culprits using the remainder of the teams' time.

There is another important side effect that results from spending too much time developing software. Brooks notes that often the product is obsolete upon (or before) completion. He refers to this as one of the woes of developing software.

Executable specifications and automatic program generation are two ideas that may provide a means to reduce the time involved in the development of software as compared with using the classical software lifecycle model. When these ideas are combined there are benefits for the entire software development process.

1.1.1 Executable Specifications As shown in Figure 1.1, the software development process has many phases, and usually the same people are not involved in each phase. The only expertise guaranteed to pass between phases is the document produced at the end of the phase. If the document is not complete, there is a possibility of introducing errors. Additionally, once the

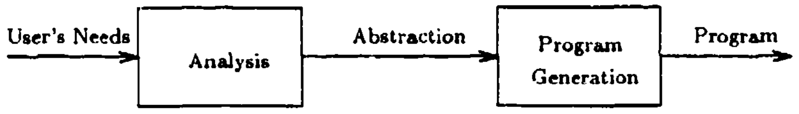


Figure 1.2. Automatic Program Generation

analysis phase is complete, the customer is usually not involved again until the end of the testing phase. This means errors or misunderstandings in the original specification are propagated through the development and not discovered until testing or operation. Errors not found until this phase are more costly to fix. Putting the specification into a form that could be executed on a computer for the customer's benefit would allow adjusting the specification to better represent the customer's needs. This fine tuning reduces the number of errors passed through the following phase by catching problems early. Also, the customer can see the system in action and is assured that the right system is being built.

1.1.2 Automatic Program Generation Lewis (12) describes a code generator, as it relates to automatic program generation, as a system that "...takes a programmer's inputs in the form of some abstraction, design, or direct interaction with the system and writes out a source program that implements the details of the application." What he sees as inputs to the system are abstractions that hide the coding details. This distinguishes code generators from tools that simply provide language templates. Using this definition allows simplification of the software development model to the representation presented in Figure 1.2. The number of phases involved in the software development lifecycle has been reduced leading to increased programmer productivity, fewer translation errors, and no direct maintenance on the code. Programmer productivity increases since the coding phase has been eliminated. Also, more products can be produced since it is only necessary to develop requirements. There are fewer errors since there are fewer people and fewer phases involved. With automatic program generation, there is no need to do maintenance on the code since the code is generated from the analysis abstractions. The abstractions are adjusted and new code is generated.

1.1.3 Combining Executable Specifications and Automatic Program Generation Executable specifications add value to the automatic generation of code. As mentioned above, finding errors early makes code production more cost effective. Being able to execute the specification gives the customer the opportunity to evaluate system operation before code is generated. Also, executable specifications combined with automatic code generation aids maintenance. In typical software maintenance environments, the documentation for the software does not match the functionality of the software. This makes it necessary to analyze the software to gain an understanding of its functionality before performing any maintenance. This activity must be performed over and over each time there is a need to change the software. This is very labor intensive, and each time a change is made the documentation bears less resemblance to the code. Using the combined methods, it is not necessary to do maintenance on the software. The specification can be changed and the new code can be generated. Now the documentation always reflects the status of the code. As mentioned by Arnold (3), if the intermediate form of the specification is available for manipulation, additional documentation can be generated. These could be items such as flow charts, dataflow diagrams, or user's manuals.

The next section describes a system that has been developed to experiment with combining the ideas of executable specifications and automatic program generation in a limited domain.

1.2 Automatic Programming Technologies for Avionics Software (APTAS)

APTAS is a system owned by Wright Laboratory which will be used as a proof-of-concept for executable specifications and automatic program generation for avionics systems. APTAS operation begins by taking in a specification for an avionics tracking system. The system selects software modules from an internal library based on rules in the system knowledge base. Once the modules have been selected, APTAS can simulate the behavior of the tracking system under development so the specification of the system can be tuned. After the desired behavior is represented, APTAS can generate the Ada code for the system.

APTAS composes modules from its library to develop the new system. However, the current APTAS library is not extensive. To get the full benefit of APTAS, the internal library must contain a large selection of tracking modules. In the past Wright Laboratory coded modules on an as-needed basis. Presently they have modules in many programming languages. Transforming the modules will be labor intensive, so automation of the transformation will be looked at to speed the task in this and similar situations.

This thesis looks at design recovery as a means for transforming existing modules to fill the library. Byrne (7) has developed a new reengineering model that outlines many of the issues involved in accomplishing the task. Both design recovery and Byrne's model are described in detail in Chapter 2.

1.3 Problem Statement

The work that needs to be accomplished to populate the library can be divided into four areas.

- Identify the intermediate language format used in the APTAS library. A template will be designed identifying important parameters and specifications for library modules.
- Characterize sample modules that need to be placed into the present library. This will entail identifying parameters required by the modules, identifying values that are returned by the modules, learning how to invoke the modules, and also identifying the modules' functionality.
- Outline a procedure that will take the module behavior and map it into the format of the template. This is also a point in the reengineering process to consider redesign of the existing modules. Ideally the mapping will produce behavior rules that APTAS can use as part of its module selection criteria.
- Implement the mapping function and test the newly derived APTAS functionality with the behavior of the original module. It is hoped that the conversion procedure can be fully automated.

1.4 Assumptions

Developing a mapping process is dependent upon being able to characterize the present APTAS knowledge base and being able to develop a template in the intermediate language accepted by APTAS. It will also be necessary to characterize modules that are to be converted. This involves capturing usage information and parameter data for each module.

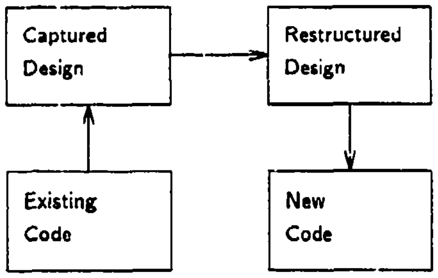


Figure 1.3. Design Recovery for Software Library Population

1.5 Scope

The research proposed here will be limited to detailing the renovation phase of Byrne's model and applying the model to the reengineering of selected tracking modules.

1.6 Summary of Current Knowledge

This thesis effort researches the possibility of using design recovery as a method for software library population. This process is presented in Figure 1.3. Beginning with existing source code, a method will be developed to capture and reconstruct a design representation. Two important issues in design recovery are determining design decisions and representing the design. Design decisions can be used to restructure the recovered design. This can be done to increase understandability, efficiency, and maintainability of the software and the design. A good representation choice will also aid in understanding and make conversion to the new system easier. Also considered at this point is adding features to the recovered design. Once the design is finalized, it must be converted to the new form. Since APTAS will generate new systems utilizing its software library, it is necessary to produce a new module in a format accepted by APTAS.

This research will not be concerned with changing the functionality of the existing software, so the design will not be restructured. It seeks to capture the design *as-is* and represent it for use in APTAS. Since the final form is for an existing system, the final representation issues have been decided. However, the actual mapping process must be addressed. These concepts will be covered in the literature review.

To ensure that all facets of the work are covered, Byrne's (7) reengineering process model will be used. Byrne's paper outlines all of the phases required for a reengineering project. It details the analysis and planning phase and gives good criteria for determining the need for a reengineering effort. A summary of his model is presented in Section 2.2.1.

1.7 Approach/Methodology

The thesis effort will begin with a survey of the literature to uncover present design recovery, automatic program generation, and reuse efforts. This will be followed by studying the intermediate library language to understand its structure and requirements. A general study of the C language will be necessary to become familiar with the modules that will be converted. Once this is complete, a mapping process will be developed, implemented, and tested.

1.8 Materials and Equipment

A complete APTAS system is required to accomplish this research. Additionally, access to the internal intermediate language formats and specifications are critical.

1.9 Sequence of Presentation

The next chapter presents a review of current work related to this thesis effort. Chapter 3 outlines the methodology that will be used to solve the problem outlined above. Chapter 4 details implementation of the methodology and Chapter 5 summarizes the results of the research. Chapter 5 also gives recommendations for additional work in the research area.

II. Literature Review

2.1 Introduction

The research presented in this thesis looks at software design recovery for library population. To better understand what is being proposed, it is necessary to understand how this fits into the broader scheme. At the outermost level of this research effort is a system designed to automatically generate other systems (see Figure 2.1). It accomplishes the generation by taking in a specification

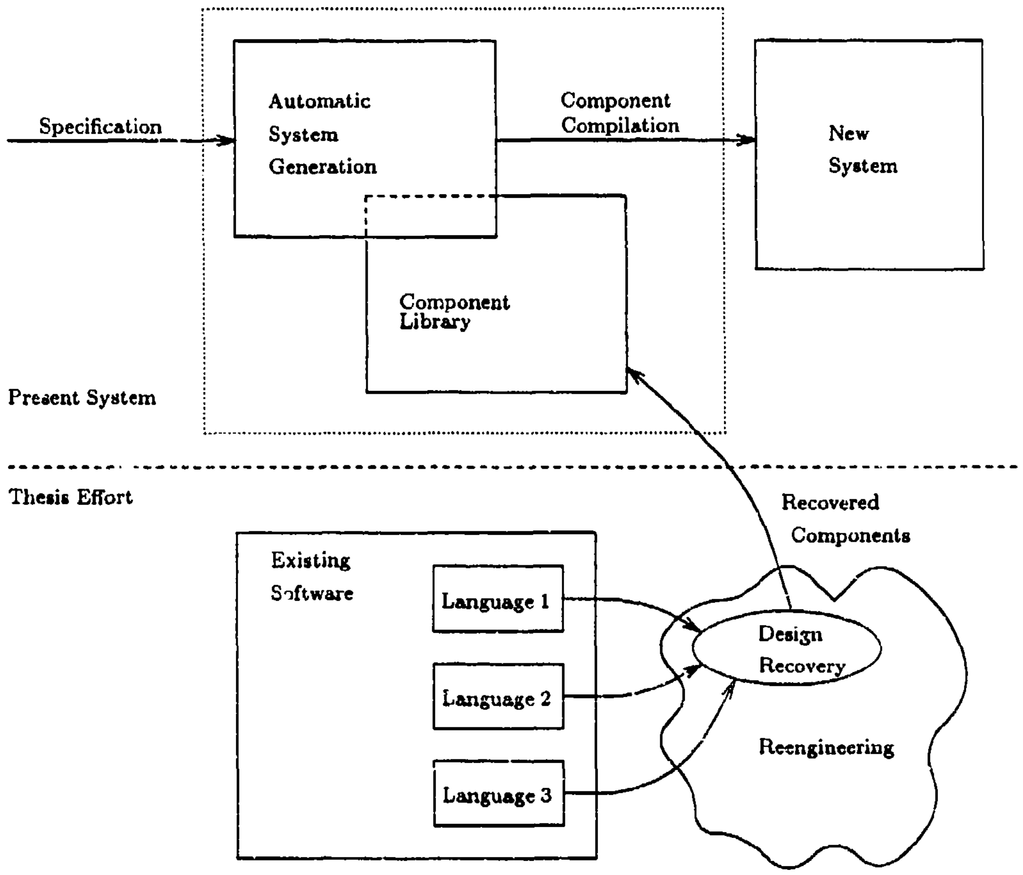


Figure 2.1. Overall Thesis Effort

and using its internal composition rules to select available modules from its internal component library that will implement the specification. Once the correct collection of components is selected, the automatic system generator can create the new system. In its present configuration, the top level system does not have many components in the internal library. This limits the types of new systems that can be produced.

Prior to APTAS, systems were constructed using the specification and manual refinement until a new coded system existed. As a result, there are existing software components but in many forms and languages. This research focused on reengineering to make existing modules available for the internal library of the top-level system. The reengineering effort examined the components as they existed with the intent of capturing their design and putting the design into a form accessible by the automatic system generator. Automatic generation of effective systems requires a large collection of library modules. Automation of the reengineering task will make best use of the automatic system generator.

This review presents current research efforts on reengineering and details one specific proposal by Byrne for modeling a reengineering project. It also covers automatic system generation with emphasis on using library components. One specific top-level system, called APTAS, and its relationship to automatic system generation and library population is defined as this system will be used as the testbed for the research. Finally, the direction of the thesis is given.

2.2 Reengineering

In any area of study confusion results when everyone has their own terms. For the field to communicate and grow, it is necessary to come up with a common set of definitions. Chikofsky and Cross (9:13-17) have baselined the field by defining the key terms associated with software reengineering. They begin with the model of the software lifecycle shown in Figure 2.2 and give the following definitions:

forward engineering the traditional process of moving from high-level abstractions and logical, implementation-independent designs to the physical implementation of a system.

reverse engineering the process of analyzing a subject system to identify the system's components and their interrelationships and to create representations of the system in another form or at a higher level of abstraction.

- **redocumentation** is a subset of reverse engineering that creates or revises a semantically equivalent representation within the same relative abstraction level. The resulting forms of representation are usually considered alternate

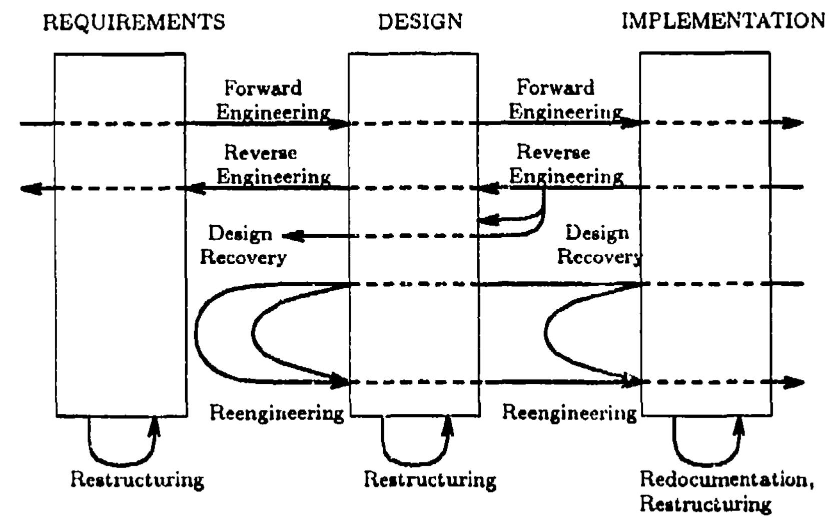


Figure 2.2. Relationship Between Terms (9)

views (e.g., dataflow, data structure, and control flow) intended for a human audience.

design recovery is a subset of reverse engineering in which domain knowledge, external information, and deduction or fuzzy reasoning are added to the observations of the subject system to identify meaningful higher level abstractions beyond those obtained directly by examining the system itself.

restructuring the transformation from one representation form to another at the same relative abstraction level, while preserving the subject system's external behavior (functionality and semantics).

reengineering also known as both renovation and reclamation, is the examination and alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form.

Throughout the remainder of this thesis use of these terms will be with the meanings given here.

2.2.1 Byrne's Model Byrne's work studies the process of software reengineering. His goal is to determine how a software reengineering project can be accomplished. He poses two questions that must be answered for any reengineering project: what information must be produced and when can this information be produced. The answers to these questions determine the information used by the process and determines the tasks and their relationships within the process. Most of the present work in reengineering emphasizes the technical aspects that must be resolved (7). It turns out that technical aspects are only one key to a successful project. Byrne has identified project management, technical work, and support as the areas that control the entire reengineering process. As shown in Figure 2.3, these processes are interwoven and must be handled together to make a

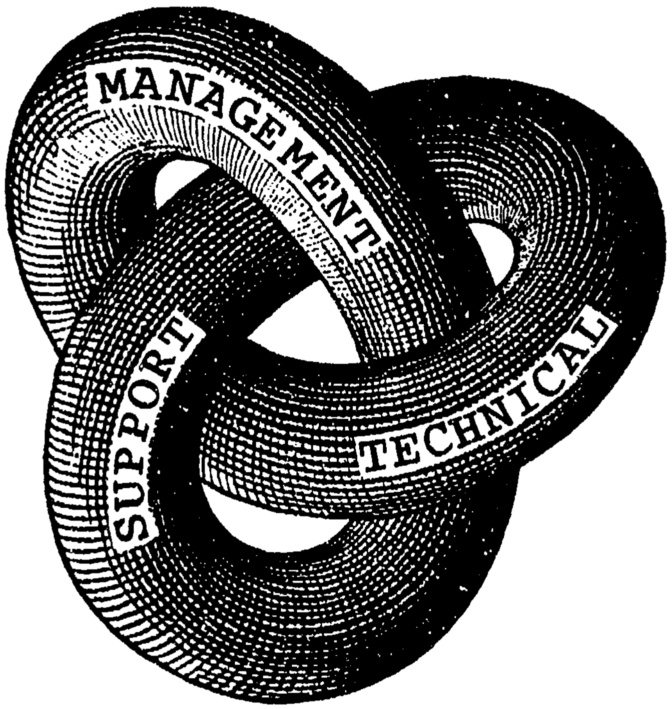


Figure 2.3. Control Area Relationships (1)

successful project. One other issue Byrne addresses is the need to specify the reengineering process unequivocally. He chose the specification language **Z** to overcome the problems that result when English is used. The following sections give an introduction to **Z** and present a general overview of Byrne's model. Also covered are the tasks for the three control areas previously mentioned and the mapping of these tasks to the phases of a reengineering project.

2.2.1.1 Introduction to **Z** **Z** is a language fc: formally specifying computer systems. It uses the mathematica' concepts of set theory and logic. For a detailed discussion, see (18). What is presented here is a few of the basics to allow an understanding of Byrne's model[1]. The basic feature of Z is the schema, and it has the following form.

```
┌─ schema − name ─────────────────────────
│ signature
├──────────
│ predicate
└─────────────────────────────────────────
```

The *schema-name* allows the schema to be included within other schemas. The *signature* identifies variable names and their types. The *predicate* describes relationships among the variables. These schemas are used to describe states, events and observations. States are mathematical structures which model a system, events are occurrences of interest, and observations are variables that can be examined before and after an event. Here are two example schemas.

```
┌─ counter ──────────────       ┌─ increment ─────────────
│ value, limit : 𝒩              │ Δcounter
├──────────                     ├──────────
│ value ≤ limit                 │ value′ = value + 1
└────────────────────────       │ limit′ = limit
                                └─────────────────────────
```

The schema on the left represents a *counter* state space. It says that there are two variables associated with the counter, its *value* and its *limit*, and these variables are natural numbers $\mathcal{N}$. The predicate says the *value* of the counter must always be less than or equal to the *limit*. The *increment* schema represents an event that changes the *counter*. Note how it uses the name of the schema on the left and the use of Δ to signify that it changes the schema. Here the predicate shows that the new counter value *value′* is the result of adding one to the present counter value *value*. The predicate also shows that there is no change in *limit*.

2.2.1.2 Model Overview By defining a model for the process of reengineering, Byrne clarifies the properties of information objects and their interrelationships without trying to capture the variety of documents involved in the process. At this level people can concentrate on why there

[1]See Appendix C for a complete list of the Z symbols used in this research

MANAGEMENT

- Define Approach
- Estimation
- Define Organizational Structure
- Define Project Procedures and Standards
- Identify Resources
- Plan System Transition
- Scheduling
- Identify Tools
- Define Acceptance Criteria
- Conflict Resolution
- Project Authorization
- Personnel Management

SUPPORT

- Configuration Management
- Quality Assurance
- Project Tracking

TECHNICAL

- Determine Motivations and Objectives
- Analyze Environments
- Collect Inventory
- Test Planning
- Target System Testing
- Documentation Planning
- Create Documentation
- Source Code Analysis
- Design Recovery
- Information Inspection
- Redesign
- Reimplementation
- Analyze New Source Code
- Acceptance Testing
- System Transition

Figure 2.4. Control Areas and Interest Items

must be a reengineering effort, what is expected of the effort, and other high-level issues. The list of tasks he defines as needing to be addressed for each of the reengineering process control areas is given in Figure 2.4.

2.2.1.3 Process Phases The reengineering phases identified by Byrne associate the interest items from Figure 2.4 with points in the reengineering process as shown in Figure 2.5. Each interest item is marked with S, M, or T to indicate whether it comes from the Support, Management, or Technical, respectively, control area. These phases cover 24 of the 30 items. Miscellaneous tasks encompass the six remaining items. These are the things that don't fit into any one phase or must be carried on throughout the project.[2]

- configuration management (S)
- quality assurance (S)
- process tracking (S)
- project authorization (M)
- personnel management (M)
- conflict resolution (M)

[2]See Byrne (7) for a complete description of the reengineering model.

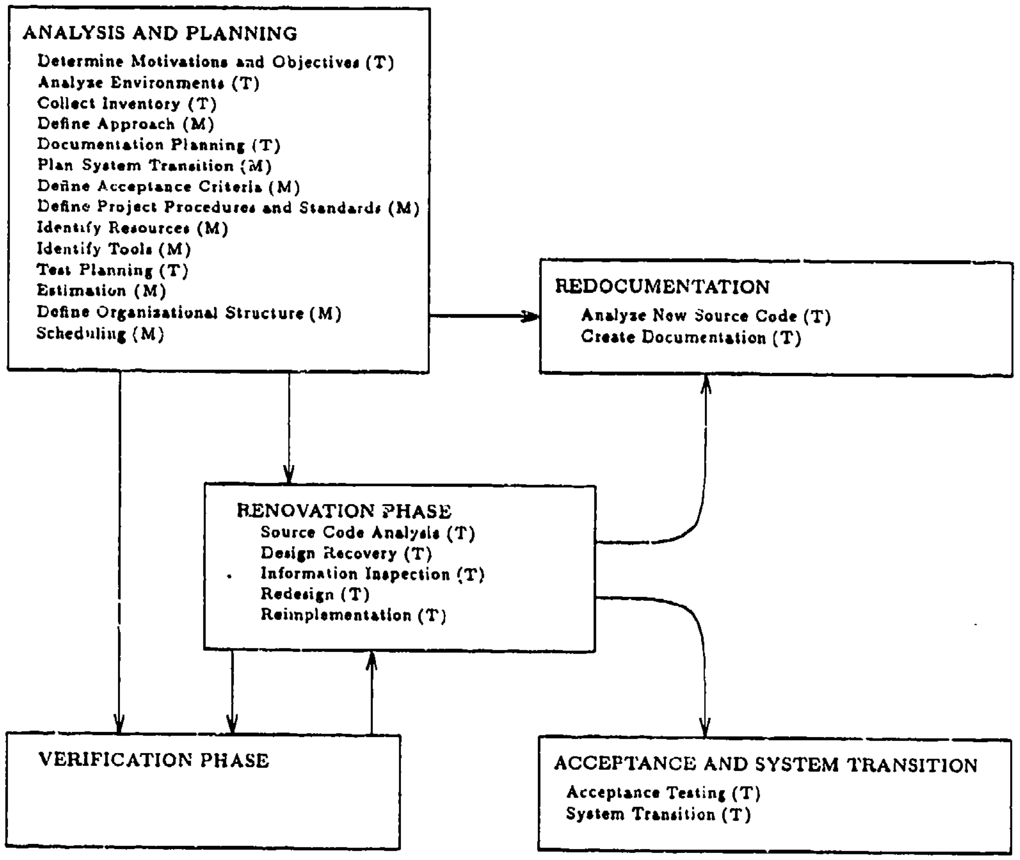


Figure 2.5. Reengineering Process Phases

A look at the **Z** specification for part of the analysis and planning phase gives more insight into the power of a formal specification language. Byrne takes the analysis and planning phase and details it in **Z**. As Figure 2.5 shows, this phase has 14 associated tasks. The details of each task look very similar in **Z**. So detailing any one task requires specifying the domain of the task, specifying the variables required to model the task, and specifying the operations available for that task. Beginning with this definition

$MOTIVATION \mathrel{\widehat{=}}$ set of all possible reengineering motivations

$OBJECTIVE \mathrel{\widehat{=}}$ set of all possible reengineering objectives

Byrne develops the following schema called the project definition that tracks and labels all reasons and goals for the reengineering project.

$$\begin{array}{|l}
\hline
\textit{DEFINITION} \\
\hline
reasons : LABEL \twoheadrightarrow MOTIVATION \\
goals : LABEL \twoheadrightarrow OBJECTIVE \\
\hline
\end{array}$$

The initial value for the project is given in the schema

$$\begin{array}{|l}
\hline
\textit{INIT\_DEFINITION} \\
\hline
DEFINITION \\
\hline
reasons = \emptyset \\
goals = \emptyset \\
\hline
\end{array}$$

The operations identified on the *DEFINITION* schema are

Add-reason	Add-goal
Delete-reason	Delete-goal
Get-reason	Get-goal
List-reasons	List-goals

and they each represent changes and operations on the state of *DEFINITION*. The schemas below show the specification for these operations with respect to *MOTIVATIONS*. The specification for

OBJECTIVE is similar. There are several new symbols in these definitions: ? signals a variable used for input; ! signals a variable used for output; and ′ signals the new value of the given variable.

$$\begin{array}{|l}
\textit{Add} - \textit{reason} \\
\hline
\Delta DEFINITION \\
m? : MOTIVATION \\
l? : LABEL \\
\hline
l? \notin \operatorname{dom} reasons \\
reasons' = reasons \cup \{l? \mapsto m?\} \\
\\
goals' = goals \\
\hline
\end{array}$$

$$\begin{array}{|l}
\textit{Delete} - \textit{reason} \\
\hline
\Delta DEFINITION \\
l? : LABEL \\
\hline
l? \notin \operatorname{dom} reasons \\
reasons' = \{l?\} \ntriangleleft reasons \\
\\
goals' = goals \\
\hline
\end{array}$$

$$\begin{array}{|l}
\textit{Get} - \textit{reason} \\
\hline
\Xi DEFINITION \\
l? : LABEL \\
m! : MOTIVATION \\
\hline
m! = reasons(l?) \\
\hline
\end{array}$$

$$\begin{array}{|l}
\textit{List} - \textit{reasons} \\
\hline
\Xi DEFINITION \\
l? : LABEL \\
m : P\{LABEL \times MOTIVATION\} \\
\hline
m = \{l : LABEL;\ m : MOTIVATION \mid reasons(l) = m\} \\
\hline
\end{array}$$

The other tasks of the analysis and planning phase were defined similarly. The first step taken was to define the domain and the variables. The second step identified the various operations that were required. And the final step specified the operations.

2.2.2 *Software Design Recovery*

Two essential steps in recovering a design are understanding what went into a design and representing this information. This section covers work that has been done in design recovery.

2.2.2.1 *Categorizing Design Decisions*

Rugaber, Ornburn, and LeBlanc (17) derived a method of characterizing design decisions by analyzing programming constructs. They note that during program development, many decisions are made. Some address the problem domain and how it should be viewed and modeled, while others address constraints imposed by the solution space, including the target machine and language. The categories they give are listed below.

composition and decomposition
encapsulation and interleaving
generalization and specialization
representation
data and procedures
function and relation

They examine a FORTRAN program and come up with the following examples as indications of design decisions.

interleaving program fragments to accomplish two calculations in a single program section.
representing structured control flow in a language that does not support them (e.g., Repeat-Until, If-Then, If-Then-Else, and Case).
interleaving by code sharing the Else part of and If-Then-Else.
data interleaving by reusing variable names for two different purposes.
generalizing interpolation schemes
variable introduction to save on repeated computation.
generalizing interval computation
representing structured control flow
program architecture

They conclude that representing design decisions will be a major factor in effective reuse. The ideal representation must be easy to construct during development and reconstruct during reverse engineering. Also, it must be formal enough to manipulate automatically and must be capable of representing all levels of design decisions.

2.2.2.2 Calling Hierarchy Another method used for design recovery begins by determining variable declarations and the respective modules (10). The next step is to find the lowest-level modules in a calling hierarchy. These are the modules that do not call other modules. This is repeated for each level until a tree-like structure has been developed representing the calling structure of the design. Lockheed has used this method to gain code understanding before grouping code segments into Ada-like structures. An additional use they found for this information is identification of components for population of a software repository.

2.3 Automatic System Generation

Present methods for generating systems have centered on two methods: generation of systems by composing components and generation of templates from a specification. Composing from components requires having a library of modules available and having a process for searching and selecting components. The template method yields a skeleton with coding details that must be completed by hand. Two applications are presented here that make use of these methods.

2.3.1 The Draco Approach Neighbors (15) researched automatic programming using an experimental prototyping system called Draco. It uses a domain language to describe programs in each different problem area. A problem area is considered a domain. Objects and operations represent analysis information about a problem domain. Analysis information states *what* is important to model in the problem. This type information is reused. Also objects and operations from one domain language can be modeled by objects and operations from other domain languages. This relationship represents different design possibilities. Design information states *how* the problem is to be modeled. Design is reused each time one of the design possibilities is used. At some level of development an executable language is needed. This is the bottom of the modeling hierarchy.

The traditional development cycle started with user and system analyst interaction to specify what the system was to do. This specification was passed to the designer to who determined how the system would accomplish the specified behavior. Draco adds two new human roles. A domain analyst examines needs and requirements of similar systems (the same problem area). This is passed to a domain designer who specifies different implementations for the various objects and operations in terms of domains already known to Draco. At this point in the development, the system analyst and user interact considering existing domains (analysis reuse). At the next stage, the designer interacts with Draco to choose a particular implementation (design reuse). The basis of the Draco work is the use of *domain analysis* to produce *domain languages* which may be *transformed* for optimization purposes and implemented by *software components*, each of which contains multiple *refinements* each of which make implementation decisions by restating the problem in other domain languages (15:565-566).

2.3.2 Issues Biggerstaff and Richter have researched the technologies that are available to address reuse. The two major areas they came up with are composition and generation. These categories were determined from the nature of the reused items. The composition group is distinguished by having atomic units that are ideally unchanged in each new application. Their example of this type of reuse is the Unix pipe that allows customizing commands by taking the output of one command and sending it through another. Their generation group is characterized by two types of patterns: code patterns and transformation patterns. Examples of these types of reuse are application generators and transformation systems. The former reuses its own internal code patterns across the generation of many systems. The latter reuses internal rules during the transformation process. In both cases it is the process that is being reused. Their assessment of reuse is that there are dilemmas that require trade offs, there are operational issues to address, and there is the issue of the level of reuse.

Within the dilemmas trade offs can be seen from many perspectives:

applicability versus payoff Technologies that are very general have a much lower payoff than systems that are narrowly focused.

component size versus reuse potential As a component grows, the payoff from reuse increases. However, the component becomes more specialized decreasing its potential for reuse.

cost of library population Usually projects are budgeted to meet short-term goals. Large initial investment for potential long-term payoffs is not seen as a viable alternative.

The operational issues they identify are finding, understanding, modifying, and composing components. Finding components includes finding exact matches as well as similar components. Without an exact match, the similar components can be used in developing a new component. Understanding a component is important to using it correctly and even more important if the component must be modified. Modifying components allows the system to evolve. They identified composing components as the most challenging because the components must be represented as distinct entities with specific characteristics and at the same time as a composition with a different characteristic.

The level of reuse can either be code or design. Code reuse has been successful in numerical computation routines. However these areas are narrow domains that are well-understood. These domains are also not rapidly changing. Design reuse is seen as an alternative, but it requires further study. If designs are represented in programming languages the designs become too specific. If they are represented very generally, they cannot be processed in machine form.

After their research they speculate that there will be very little immediate progress because of the initial investment required. Also additional research is required to overcome the design representation issues.

2.4 APTAS

As stated earlier, the research proposed here will make library modules available for a system that automatically generates programs by composing components. Figure 2.6 is an APTAS system diagram which emphasizes the interfaces presented to the user during development of an application (referred to in APTAS as a *project*). An engineer using the system begins defining a tracking application's specification in the *taxonomy summary window*. It contains of a set of text lines, each representing a form. The forms present questions using the *dynamic forms interface*. A form takes numeric, text string, exclusive choice, or checklist information. By answering the presented questions, the specification is developed. The number of entries appearing in the summary window increases, reflecting the effects form selections have had in pruning the taxonomy tree toward a specific architecture.

When the forms are complete, they are submitted to the *architecture generator*. The generated architecture is presented in the *graphical user interface* (GUI). It presents a graphical representation of the generated architecture using components that can be edited to provide the specification's details before code synthesis takes place. There are four types of icons used in the GUI: a box represents a module; a circle represents the communications interface of a module; a diamond represents the interface function of a module; and a triangle represents a parameter of a module. There are also lines representing relations between the modules. A sample module is presented in Figure 2.7.

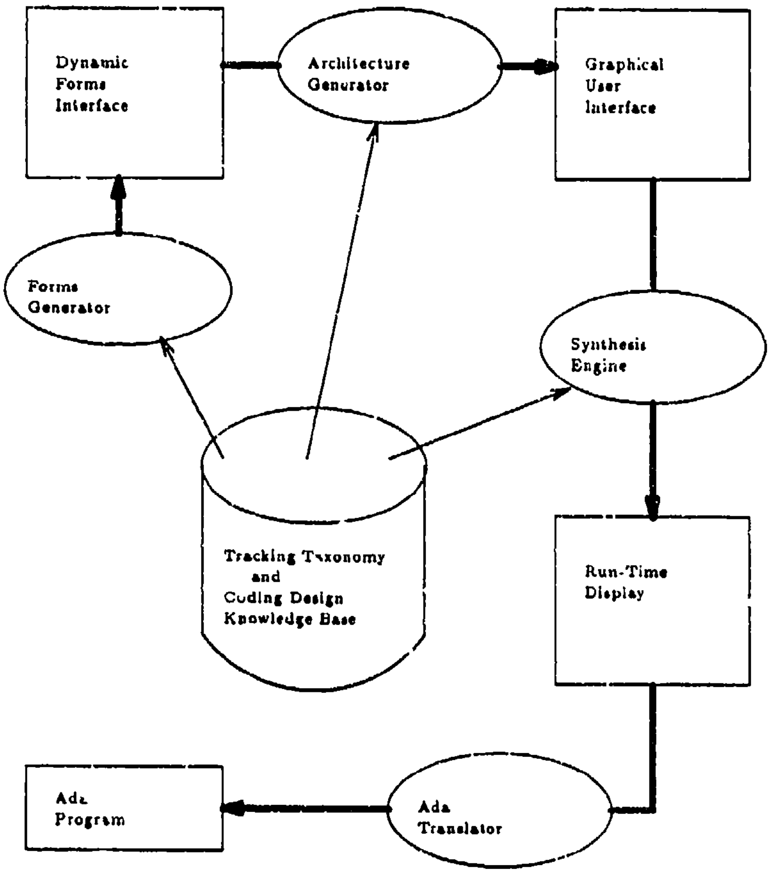


Figure 2.6. APTAS Organizational Diagram (13)

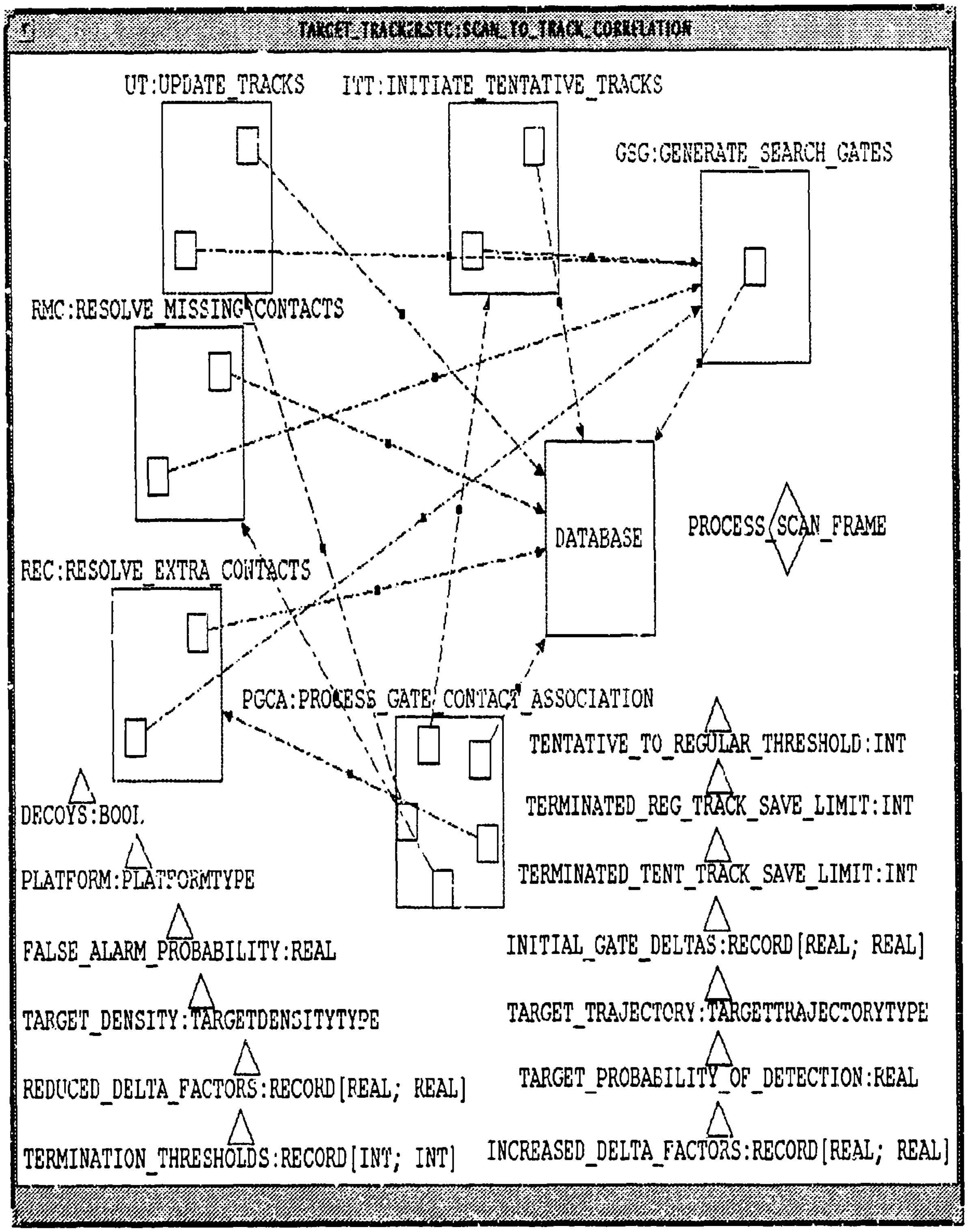
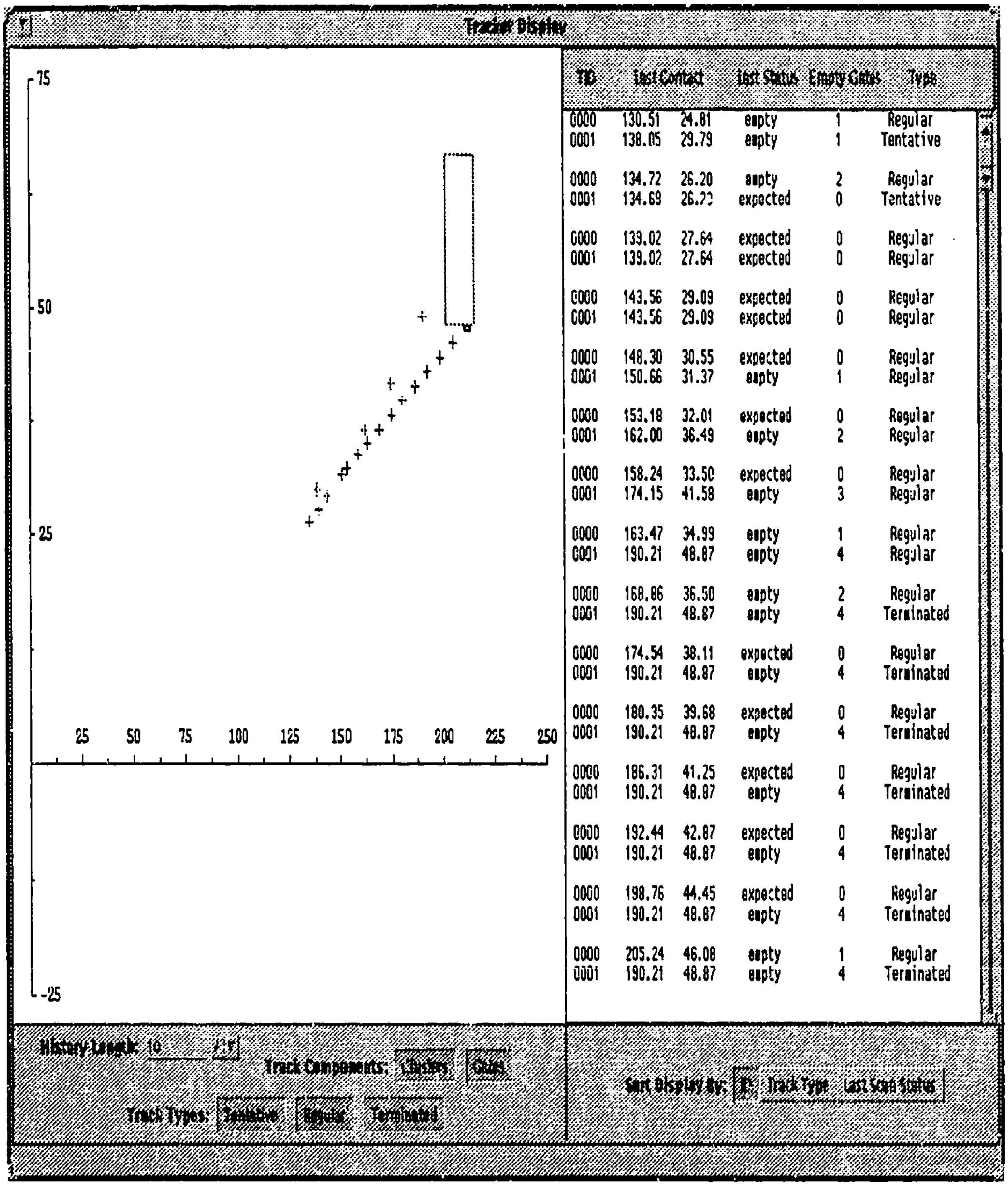

TARGET_TRACK:STC:SCAN_TO_TRACK_CORRELATION
UT:UPDATE_TRACKS
ITT:INITIATE_TENTATIVE_TRACKS
GSG:GENERATE_SEARCH_GATES
RMC:RESOLVE_MISSING_CONTACTS
DATABASE
PROCESS_SCAN_FRAME
REC:RESOLVE_EXTRA_CONTACTS
PGCA:PROCESS_GATE_CONTACT_ASSOCIATION
TENTATIVE_TO_REGULAR_THRESHOLD:INT
TERMINATED_REG_TRACK_SAVE_LIMIT:INT
TERMINATED_TENT_TRACK_SAVE_LIMIT:INT
INITIAL_GATE_DELTAS:RECORD[REAL; REAL]
TARGET_TRAJECTORY:TARGETTRAJECTORYTYPE
TARGET_PROBABILITY_OF_DETECTION:REAL
INCREASED_DELTA_FACTORS:RECORD[REAL; REAL]
DECOYS:BOOL
PLATFORM:PLATFORMTYPE
FALSE_ALARM_PROBABILITY:REAL
TARGET_DENSITY:TARGETDENSITYTYPE
REDUCED_DELTA_FACTORS:RECORD[REAL; REAL]
TERMINATION_THRESHOLDS:RECORD[INT; INT]

Figure 2.7. GUI Representation of a Module

When the specification has been completed in the graphical user interface, an implementation may be generated in CIDL by pushing the *Synthesize* button on the APTAS system control panel. CIDL is a high level system design language developed at the Lockheed Software Technology Center (LSTC) as part of LSTC's Software Synthesis project. Once the CIDL code has been generated, an equivalent Ada implementation can be generated by pushing the *Translate* button on the APTAS system control panel. The behavior of the generated CIDL and Ada tracker implementations may be tested by invoking the *Execute* button's menu from the APTAS system control panel, then selecting either *Run* CIDL or *Execute Ada*. When a selection is made, the code will begin executing, and a window will be displayed showing the output of the tracker. The output is simultaneously written to files for future analysis and/or utilization of the Run-Time Display program. The data generated from the executing tracker may be presented in a visual display (shown in Figure 2.8). If the user is not satisfied with the test results, he/she may return to the GUI or *taxonomy summary window* to modify the specification and repeat the synthesis and test processes. The *tracking taxonomy and coding design knowledge base* is used to support multiple phases of the specification and synthesis process.

2.5 APTAS Library Population

Extending the tracking taxonomy and coding knowledge base entails writing CIDL implementations of primitive modules, rules which determine when the primitive is appropriate for a given application, and the questions to present to the user which will elicit the information needed to evaluate those rules. The CIDL module construct, used to define the reusable primitive software components of the APTAS knowledge base, defines a new type which encapsulates a set of types, declarations, and functions. A module type declaration includes up to four sections: parameters, interface, structure, and behavior. Parameters provide the *generic* character of modules. The exact properties of each instantiation of the module type depend on the parameter values provided when the instance was created. A sample module is given in Figure 2.9. The interface describes which components of the module are accessible outside the module. The structure section contains the local declarations. The behavior section describes the processing which takes place each time the module type is instantiated. Instantiation is performed by a call to the module creation function which is generated when the module type is compiled. Adding a primitive to the taxonomy requires


Tracker Display
75
50
25
-25
25
50
75
100
125
150
175
200
225
250
History Length: 10
Track Components: Clusters Gates
Track Types: Tentative Regular Terminated
Sort Display By: Track Type Last Scan Status

```
TID    Last Contact      Last Status  Empty Gates   Type
0000   130.51   24.81    empty        1             Regular
0001   138.05   29.79    empty        1             Tentative

0000   134.72   26.20    empty        2             Regular
0001   134.69   26.7[illegible]    expected     0             Tentative

0000   139.02   27.64    expected     0             Regular
0001   139.02   27.64    expected     0             Regular

0000   143.56   29.09    expected     0             Regular
0001   143.56   29.09    expected     0             Regular

0000   148.30   30.55    expected     0             Regular
0001   150.66   31.37    empty        1             Regular

0000   153.18   32.01    expected     0             Regular
0001   162.00   36.49    empty        2             Regular

0000   158.24   33.50    expected     0             Regular
0001   174.15   41.58    empty        3             Regular

0000   163.47   34.99    empty        1             Regular
0001   190.21   48.87    empty        4             Regular

0000   168.86   36.50    empty        2             Regular
0001   190.21   48.87    empty        4             Terminated

0000   174.54   38.11    expected     0             Regular
0001   190.21   48.87    empty        4             Terminated

0000   180.35   39.68    expected     0             Regular
0001   190.21   48.87    empty        4             Terminated

0000   186.31   41.25    expected     0             Regular
0001   190.21   48.87    empty        4             Terminated

0000   192.44   42.87    expected     0             Regular
0001   190.21   48.87    empty        4             Terminated

0000   198.76   44.45    expected     0             Regular
0001   190.21   48.87    empty        4             Terminated

0000   205.24   46.08    empty        1             Regular
0001   190.21   48.87    empty        4             Terminated
```

Figure 2.8. Run-Time Display

```
module Sensor_Model

   Parameters

      sensor: SensorType;
      target_specs: sequence(GenericTargetSpec);
      perturbation_factor: real;
      iterations: int;
      scan_frame_out: event(GenericScanFrame)

   Structure

      loop_counter: store(int);
      sensor_time: store(real);


   Behavior

      loop_counter := 0;
      sensor_time := 0.0;

EndModule;
```

Figure 2.9. Sample CIDL Module

adding the appropriate forms information and module selection criteria. These steps are performed by first determining where the new primitive module fits into the existing taxonomy, determining the conditions under which it should be selected for a particular application, and adding the appropriate entry to the list of available modules. The next step is to determine the appropriate forms and/or questions for existing forms required to solicit the information needed to evaluate those conditions.

2.6 Review Summary

This chapter has presented current work in the field of design recovery and automatic program generation. There was also a summary of Byrne's reengineering model. This research will use the model as a framework to capture all of the issues that need to be addressed in outlining a reengineering project. The methods of design recovery and automatic program generation will be examined for a solution to the problem of populating the APTAS library.

III. Methodology

This chapter outlines the two-step approach selected to solve the library population problem. The first step entails using Byrne's reengineering model to guide the project. As discussed earlier, a reengineering project involves technical issues as well as support and management issues. Byrne's model was chosen because it deals with all of these issues. With his complete description of the analysis and planning phase, Byrne has a good foundation for determining the need for a reengineering effort and the resources that will be required to complete the project. Additionally, the amount of detail in the model will ensure that all issues are addressed and tracked. A major part of the first step elaborated the renovation phase of the reengineering model, in Byrne's notation, since it was not developed in detail by Byrne. The second step was to reengineer the existing software in the new form. This step applied the concepts of the first step serving as a proof-of-concept for the model and for using design recovery as a solution to the library population problem. The approach taken in this chapter is to demonstrate how to use Byrne's model by applying his specification of the analysis and planning phase to the library problem, to develop the renovation phase, and finally to apply the model to the library problem.

3.1 Analysis and Planning Phase

Byrne's original work in this phase was done using **Z**. This language is based on formal logic and set theory. With this very mathematical foundation, it reduces the ambiguity in the resulting model. This research will continue using **Z** to maintain a low level of ambiguity. An added benefit will be easier enactment of the model should that become necessary. REFINE^{TM} is an example of a programming language that could be used to enact this model since it to is based on set theory and formal logic.

The previously defined Analysis and Planning phase is shown in Figure 3.1. The tasks have been identified by Byrne and represent management (M), technical (T), or support (S) issues. Each task has associated characteristics that must be tracked. To follow these characteristics operations are identified for each task. The characteristics are represented as a set of partial functions. This means there is a mapping from a name for a characteristic and the associated entry for the particular characteristic. Since all of these tasks are represented as sets, they have common operations that

ANALYSIS AND PLANNING

- Determine Motivations and Objectives (T)
- Analyze Environments (T)
- Collect Inventory (T)
- Define Approach (M)
- Documentation Planning (T)
- Plan System Transition (M)
- Define Acceptance Criteria (M)
- Define Project Procedures and Standards (M)
- Identify Resources (M)
- Identify Tools (M)
- Test Planning (T)
- Estimation (M)
- Define Organizational Structure (M)
- Scheduling (M)

Figure 3.1. Analysis and Planning Phase

can be performed on them. These operations are add an item to the set, delete an item from the set, list the items in the set, and get an item from the set for modification. Here is an example using the task Determine Motivations and Objectives as it applies problem of library population.

This task tracks all of the motivations and objectives for a project. The Z schemas were identified in 2.2.1.3 to add, delete, get, and list all of the reasons and goals for any project. This particular project begins with the DEFINITION schema showing reasons and goals as empty sets.

DEFINITION

$motivations(T) : \emptyset$

$objectives(T) : \emptyset$

The *T* signifies that these are technical tasks as previously defined. As motivations and objectives are identified for the project, the add function is applied to each of these tasks to document this information. The resulting schema for the library problem is shown here.

DEFINITION

motivations(T) : {

($mot1 \mapsto$ (there are existing routines that implement functions that are also required in the new system)),

($mot2 \mapsto$ (the new system does not have sufficient routines to be effective)),

($mot3 \mapsto$ (the existing routines are implemented in many programming languages)),

($mot4 \mapsto$ (the existing routines are spread over many computers)),

($mot5 \mapsto$ (the existing routines are ad hoc; appear to be only home grown utilities)),

($mot6 \mapsto$ (there is incentive to take advantage of existing routines in a new technology that generates Ada code from tracking algorithms))

}

objectives(T) : {

($obj1 \mapsto$ (to make additional routines available for automatic system generation)),

($obj2 \mapsto$ (to improve software system maintainability)),

($obj3 \mapsto$ (to convert the existing library to a single language that can be used to generate Ada)),

($obj4 \mapsto$ (to port the existing software to a single system))

}

Application of the **Z**-defined operators to the other tasks is similar. As pointed out in (8), a major output of the analysis part of this phase is the current status of the system. The planning part of this phase outlined the management issues including identifying the scope of the work, the required resources, milestones, and establishing a schedule. The phys[illegible]al outputs of this phase are an overall project plan, a plan for the other phases, and the existing [illegible] documentation. The renovation phase is one of the phases that follows analysis and plan[illegible]

3.2 System Renovation Phase

At this phase the existing system is transformed into the target sys[illegible] [illegible]his transformation follows the steps outlined in (8). There are five tasks used to accomplish these steps.

- Source Code Analysis
- Design Recovery
- Information Inspection
- Redesign
- Reimplementation

This section details these tasks using **Z**. This phase begins with some of the outputs from the analysis and planning phase. Additional inputs required for this phase are existing standards.

These standards are items required of all projects. Considering the first two tasks together forms reengineering as defined earlier. The items produced by these tasks include a data dictionary and a cross reference of all files and variables. Since this task is starting with source code, the initial issue would consider capturing a software design-level representation.

3.2.1 Background In his original definition of the model, Byrne defined some global sets that contained items used in every phase of the project. This section reviews these definitions since they will be used as part of the description of the schemas and operations of the tasks found in the renovation phase. Dates, names, labels, and conditions are used extensively throughout the project. Dates are associated with task starts and stops as well as phase starts and stops for example. Names are assigned to personnel, files, and tasks. Labels could be associated with steps in a procedure or items in a collection. Finally, conditions are used to signify whether or not things can occur. To keep track of all of these, four sets have been created.

$DATE \mathrel{\widehat{=}}$ **set of all valid dates**
$NAME \mathrel{\widehat{=}}$ **set of names**
$LABEL \mathrel{\widehat{=}}$ **set of all labels**
$CONDITION \mathrel{\widehat{=}}$ **set of all conditions**

Similar to the idea of sets to represent common items is the need to represent lists of characteristic. For this Byrne identified the PROPERTY_LIST. It is used to track named properties and and the associated values. He starts by identifying all properties and all values as sets.

$PROPERTY \mathrel{\widehat{=}}$ **set of all properties**
$VALUE \mathrel{\widehat{=}}$ **set of all property values**

Once an item has been identified as having many properties that need to be tracked, a property list can be created associating a collection of property names with a collection of property values. Here PROP_NAME and PROP_LIST are specified and the initial value of the property list is given.

$PROP\_NAME \mathrel{\widehat{=}}$ **set of all property names**

$PROP\_NAME \subset NAME$

$PROP\_LIST : PROP\_NAME \nrightarrow VALUE$
$INITIAL\_PROP\_LIST = \emptyset$

For the general property list, referred to as *pl* below, the operations add, update, delete, get, and list are defined. These operations can be instantiated for any list. A Δ before the list name indicates that the list changes after an operation, and a Ξ before the name indicates that the operation does not cause a change in the list composition. The add, delete, and update operations cause changes in the list, while the get and list do not cause changes. The general form of these operations are given here.

Add – Property

$$\begin{array}{|l}
\Delta PROP\_LIST \\
p? : PROP\_NAME \\
v? : VALUE \\
\hline
p? \notin \operatorname{dom} pl \\
pl' = pl \cup \{p? \mapsto v?\}
\end{array}$$

Update – Property

$$\begin{array}{|l}
\Delta PROP\_LIST \\
p? : PROP\_LIST \\
v? : VALUE \\
\hline
p? \in \operatorname{dom} pl \\
pl' = pl \oplus \{p? \mapsto v?\}
\end{array}$$

Delete – Property

$$\begin{array}{|l}
\Delta PROP\_LIST \\
p? : PROP \\
\hline
p? \in \operatorname{dom} pl \\
pl' = \{p?\} \lhd pl
\end{array}$$

Get – Property

$$\begin{array}{|l}
\Xi PROP\_LIST \\
p? : PROP\_NAME \\
v! : VALUE \\
\hline
p? \in \operatorname{dom} pl \\
v! = pl(p?)
\end{array}$$

List – Properties

$$\begin{array}{|l}
\Xi PROP\_LIST \\
list! : P\{PROP\_NAME \times VALUE\} \\
\hline
list! = \{p : PROP\_NAME;\ v : VALUE \mid pl(p) = v\}
\end{array}$$

Knowing the basic definitions will aid in understanding the definitions to follow. The first task in the renovation phase is source code analysis.

3.2.2 Source Code Analysis The input to this task is the existing software and outputs are source code information and a data dictionary. To capture the source code information we assume it is contained in one or more files having similar properties. Analysis proceeds from the level of identifying files down through identifying procedures and functions, subroutines, and variables. The reason for following this pattern is that it structures the analysis, and it follows the pattern used by people going from the general to the specific. The collection of files is defined as

$FILES \mathrel{\widehat{=}}$ set of all possible project files

Each file has a name that is a member of $NAME$ previously identified. The specific names used for files will be denoted FILE_NAME.

$FILE\_NAME \mathrel{\widehat{=}}$ set of all possible file names

$FILE\_NAME \subset NAME$

Each file has many properties associated with it. For files that are to be converted to the new system information that must be collected includes the location of the file, the type file (e.g. input data or binary output), the language used in the file, the names of files that use it, the names of other files that it uses, and the contents of the file. Since this information should be collected on each file a property list is setup to ensure complete collection of information for each file.

$FILE\_PROPERTIES \mathrel{\widehat{=}}$ Predefined set of all file properties

$FILE\_PROPERTIES \subset PROP\_LISTS$

At this point in the reengineering effort, it is necessary to track all of the files needed by the project. The schema PROJECT_FILES is defined to track these items.

$PROJECT\_FILES$

$file\_list : \mathbb{F}\, FILE\_NAME$

$file\_info : FILE\_NAME \twoheadrightarrow PROP\_LISTS$

Once the files have been identified, file contents can be analyzed. Within the files items that are expected are procedures, functions, subroutines, and variables. Each of these also have properties associated with them. Procedures, functions, and subroutines are similar in nature and require tracking of information such as the item name, the functionality provided by the item, parameters required by the item, expected results, the type item, where it is declared, and where the item is used. Variables require tracking information such as the name of the variable, its type, where it is declared, where it is used, and its purpose. Two new collections are identified to track the names of these items

$ROUTINES \mathrel{\widehat{=}}$ **set of all possible project procedures, functions, and subroutines**
$VARIABLES \mathrel{\widehat{=}}$ **set of all possible project variables**

and property lists are established to track the associated properties.

$ROUTINE\_PROPERTIES \mathrel{\widehat{=}}$ **Predefined set of all routine properties**
$VARIABLE\_PROPERTIES \mathrel{\widehat{=}}$ **Predefined set of all variable properties**

$ROUTINE\_PROPERTIES \subset PROP\_LISTS$
$VARIABLE\_PROPERTIES \subset PROP\_LISTS$

Finally, schemas are created to characterize the routines and variables.

$PROJECT\_ROUTINES$
$routine\_list : \mathbb{F}\, ROUTINE\_NAME$
$routine\_info : ROUTINE\_NAME \twoheadrightarrow PROP\_LISTS$

$PROJECT\_VARIABLES$
$variable\_list : \mathbb{F}\, VARIABLE\_NAME$
$variable\_info : VARIABLE\_NAME \twoheadrightarrow PROP\_LISTS$

The data dictionary produced in forward engineering defines all of the data used in the system being developed. Reengineering using the above definitions recovers all of the original data definitions from the existing system. In addition it identifies the incidental variables, procedures, subroutines, and functions and shows all of the relationships between these items. Once the files, routines, and variables have been identified, it is time to proceed to the next level in reengineering the system: Design Recovery.

3.2.3 Design Recovery This task of the renovation phase adds domain knowledge and external information as pointed out in the definition of design recovery. A major portion of this task is providing the information that links the items identified in the previous task. This task may provide additional information for the present property lists or identify additional properties that need to be included in the lists. Also identified are more of the *what* is being accomplished by the system that is being reengineered.

Something that surfaces at this point in the reengineering project is how to represent the recovered design. There are many tools that can be used such as structure charts, transition diagrams, and program description languages. It does not seem that any one tool is overall better than any other. However, choosing a tool based on the desired outcome does help. A hypertext type tool (5) that allows multiple views of the same information seems to be ideal. This would allow viewing the information at a level of abstraction on par with the task at hand. The specific tool that is used to capture design information is something that should be outlined in the standards of the organization involved in the reengineering project. What is defined here is a way to track the products for a particular project.

Individual products will have names to distinguish them from other items. It is also necessary to track their location and details. Here details refers to the composition of the product whether they are diagrams or descriptions. Since any project can have multiple design products, a method is needed to track items associated with a particular project. The definitions necessary to carry out these tasks are identified below.

$DESIGN\_PRODUCT \mathrel{\hat{=}}$ **set of all possible design products**
$PROJECT \mathrel{\hat{=}}$ **set of all possible reengineering projects**

$DESIGN\_PRODUCT\_NAME \mathrel{\hat{=}}$ **set of all possible design product names**
$PROJECT\_NAME \mathrel{\hat{=}}$ **set of all possible project names**

$DESIGN\_PRODUCT\_NAME \subset NAME$
$PROJECT\_NAME \subset NAME$

```
_PROJECT_DESIGN______________________________________
| project_name : F PROJECT
| project_info : PROJECT ⇸ DESIGN_PRODUCT_NAME
|_____________________________________________________
```

The recovered design is now ready for passing to the next task in the renovation phase.

3.2.4 Information Inspection and Redesign In the information inspection task the details of how to achieve the objectives are addressed. The output produced is a plan for changes to the recovered design to get the new design. This plan will probably be in the form of steps that need to be accomplished. Since the recovered design has created artifacts similar to those used in forward engineering, the same methods could be used to plan redevelopment of the system.

The redesign task of the renovation phase allows for adjustments to the design to aid future maintenance of the system. Also, this is the point in the project where improvements and new requirements could be added to the system. As pointed out in (8), there is an iterative relationship between these two tasks. Changes to the design require additional planning which may result in additional changes to the design. Therefore, a method is needed to track all of the changes that occur during these two tasks.

The goal of this project is to evaluate the concept of library population. Since the modules that will be transformed are assumed to exhibit the required behavior, redesign will not play a part in this reengineering effort. This phase of the reengineering model will not be used for this particular project. However, if population seems viable, this step must be reexamined.

3.2.5 Reimplementation Reimplementation is the phase that actually produces the new system. Once progress has reached this phase, traditional forward engineering methods can continue to be used. What is also necessary is a method to ensure that all required recovered behavior is implemented. Part of this task is unit testing. With the test plans that have been developed and the behavior that has been noted in the previous tasks, this should make verification of proper behavior easier.

Now that the tasks have been outlined for the library population problem, they can be put to the task at hand. The next section looks at developing the items called for in a reengineering project.

3.3 Library Population

At this point in the research, it is necessary to obtain actual modules that need to be transformed. Appendix D lists the first module selected for the transformation. It is not too large and is used in several places in the new system. All of the source code is contained in one file, however, the module produces several output files and has several variables and subroutines. What follows is a description of the information recovered as a result of applying the various **Z** definitions.

3.3.1 Source Code Analysis This task is started by identifying the files associated with this task. Since this task will analyze the existing source code, properties are then identified to guide the collection of information. The first level of analysis is with the files involved. These are the properties identified with information to be gathered about each file in the collection of files.

$$FILE\_PROPERTIES \mathrel{\widehat{=}} \{location, type, language, uses\_files, used\_in, contents\}$$

After the specific properties are identified, this is compared with the list of related files and produces the following schema.

```
PROJECT_FILES
file_list : {
    EPSILON.PRN, KALMAN.PL, KALMAN.PLT, P11.PRN, P22.PRN, XEST.PRN
    XMEAS.PRN, XNOISE.PRN, XTILDE.PRN, XTRUTH.PRN, YEST.PRN, YMEAS.PRN
    YNOISE.PRN, YTURTH.PRN, ZXNOISE.PRN, ZYNOISE.PRN, proj1.for
}
file_info : {
    (EPSILON.PRN ↦ (same directory, output, ASCII, N/A, N/A, data to be printed)),
    (KALMAN.PL ↦ (same directory, output, ASCII, N/A, N/A, data to be printed)),
    (KALMAN.PLT ↦ (same directory, output, ASCII, N/A, N/A, data to be printed)),
    (P11.PRN ↦ (same directory, output, ASCII, N/A, N/A, data to be printed)),
    (P22.PRN ↦ (same directory, output, ASCII, N/A, N/A, data to be printed)),
    (XEST.PRN ↦ (same directory, output, ASCII, N/A, N/A, data to be printed)),
    (XMEAS.PRN ↦ (same directory, output, ASCII, N/A, N/A, data to be printed)),
    (XNOISE.PRN ↦ (same directory, output, ASCII, N/A, N/A, data to be printed)),
    (XTILDE.PRN ↦ (same directory, output, ASCII, N/A, N/A, data to be printed)),
    (XTRUTH.PRN ↦ (same directory, output, ASCII, N/A, N/A, data to be printed)),
    (YEST.PRN ↦ (same directory, output, ASCII, N/A, N/A, data to be printed)),
    (YMEAS.PRN ↦ (same directory, output, ASCII, N/A, N/A, data to be printed)),
    (YNOISE.PRN ↦ (same directory, output, ASCII, N/A, N/A, data to be printed)),
    (YTURTH.PRN ↦ (same directory, output, ASCII, N/A, N/A, data to be printed)),
    (ZXNOISE.PRN ↦ (same directory, output, ASCII, N/A, N/A, data to be printed)),
    (ZYNOISE.PRN ↦ (same directory, output, ASCII, N/A, N/A, data to be printed)),
    (proj1.for ↦ (same directory, main routine, FORTRAN, N/A, N/A, subroutines and variables))
}
```

Source code analysis continues by looking into the various files. In this instance, the only file that needs to be examined is *proj1.for* since all of the other files are produced by executing this file. This portion of the analysis is looking for procedures, functions, subroutines, and variables. Again, the first step is defining the properties that need to be collected for these items. Something that comes up at this point is the hierarchical manner of declaring procedures and variables. They can be declared in one place and used in another. Usage can be at a different level than the declaration. This requires the usage level to also be captured. To accomplish this, usage will be represented as *file/procedure/.../procedure* until the proper level is reached. The first level is represented by the *file* in which usage occurs. *Procedures* or *subroutines* are added for each corresponding level. In defining this information, scoping rules are necessary. The procedure used for scoping is to identify routines and variables that are one level down from the routine of interest. The following properties are identified for collection.

$ROUTINE\_PROPERTIES \mathrel{\widehat{=}}$ {function, parameters, results, type, declared_in, used_in}
$VARIABLE\_PROPERTIES \mathrel{\widehat{=}}$ {type, declared_in, used_in, purpose}

Collecting the variable and routine information results in the following schemas.

```
__PROJECT_ROUTINES_________________________________________
| routine_list : {main, noise, mtxmul, mtxadd, mtxsub, mtxzro, mtxtrp, mtxinv, idemtx}
| routine_info : {
|      (main ↦ (kalman filter implementation,N/A,creates files,procedure,/proj1.for,N/A)),
|      (noise ↦ (generates gaussian noise,(xmean, variance, rndmn, n),
|            matrix initialized with noise,subroutine,/proj1.for,/proj1.for/main)),
|      (mtxmul ↦ (matrix multiplication, (a,b,c,n1,n2,n3), a*b,subroutine,/proj1.for,/proj1.for/main)),
|      (mtxadd ↦ (matrix addition, (a,b,c,n1,n2),a+b,subroutine,/proj1.for,/proj1.for/main)),
|      (mtxsub ↦ (matrix subtraction,(a,b,c,n1,n2),a-b,subroutine,/proj1.for,/proj1.for/main)),
|      (mtxzro ↦ (matrix zero, (a,n1,n2), a,subroutine, /proj1.for,/proj1.for/main)),
|      (mtxtrp ↦ (matrix transpose, (a,b,1n,n2), b,subroutine,/proj1.for,/proj1.for/main)),
|      (mtxinv ↦ (matrix inverse, (a,ainv,b,kc,is), (ainv,is),subroutine,/proj1.for,/proj1.for/main)),
|      (idemtx ↦ (identity matrix, (a,n), a,subroutine, /proj1.for,/proj1.for/main))
| }
|___________________________________________________________
```

```
PROJECT_VARIABLES
variable_list : {
        K, C, I1, I2, IS, IOPT, NGR, NPT, X, Y, WX, WY, XHATN, XHATOLD,
        VX, VY, Z, XHAT, ZHAT, NU, P, F, FT, R, PN, TEMP1, TEMP2, Q, H, S,
        HT, SI, W, WT, XTILDE, XTILDET, EPSILON, Graph, PI, Nameg
}
variable_info : {
        (K ↦ (integer, /proj1.for/main, /proj1.for/main, loop counter)),
        (C ↦ (integer, /proj1.for/main, /proj1.for/main, sample counter)),
        (I1 ↦ (integer, /proj1.for/main, /proj1.for/main, seed)),
        (I2 ↦ (integer, /proj1.for/main, /proj1.for/main, seed)),
        (IS ↦ (integer, /proj1.for/main, /proj1.for/main, matrix singular flag)),
        (IOPT ↦ (integer, /proj1.for/main, /proj1.for/main, purp)),
        (NGR ↦ (integer, /proj1.for/main, /proj1.for/main, purp)),
        (NPT ↦ (integer, /proj1.for/main, /proj1.for/main, purp)),
        (X ↦ (array of reals, /proj1.for/main, /proj1.for/main, purp)),
        (Y ↦ (array of reals, /proj1.for/main, /proj1.for/main, purp)),
        (WX ↦ (array of reals, /proj1.for/main, /proj1.for/main, purp)),
        (WY ↦ (array of reals, /proj1.for/main, /proj1.for/main, purp)),
        (XHATN ↦ (array of reals, /proj1.for/main, /proj1.for/main, purp)),
        (XHATOLD ↦ (array of reals, /proj1.for/main, /proj1.for/main, purp)),
        (VX ↦ (array of reals, /proj1.for main, /proj1.for/main, purp)),
        (VY ↦ (array of reals, /proj1.for/main, /proj1.for/main, purp)),
        (Z ↦ (array of reals, /proj1.for/main, /proj1.for/main, purp)),
        (XHAT ↦ (array of reals, /proj1.for/main, /proj1.for/main, purp)),
        (ZHAT ↦ (array of reals, /proj1.for/main, /proj1.for/main, purp)),
        (NU ↦ (array of reals, /proj1.for/main, /proj1.for/main, purp)),
        (P ↦ (array of reals, /proj1.for/main, /proj1.for/main, purp)),
        (F ↦ (array of reals, /proj1.for/main, /proj1.for/main, purp)),
        (FT ↦ (array of reals, /proj1.for/main, /proj1.for/main, purp)),
        (R ↦ (array of reals, /proj1.for/main, /proj1.for/main, purp)),
        (PN ↦ (array of reals, /proj1.for/main, /proj1.for/main, purp)),
        (TEMP1 ↦ (array of reals, /proj1.for/main, /proj1.for/main, temporary matrix)),
        (TEMP2 ↦ (array of reals, /proj1.for/main, /proj1.for/main, temporary matrix)),
        (Q ↦ (array of reals, /proj1.for/main, /proj1.for/main, purp)),
        (H ↦ (array of reals, /proj1.for/main, /proj1.for/main, purp)),
        (S ↦ (array of reals, /proj1.for/main, /proj1.for/main, purp)),
        (HT ↦ (array of reals, /proj1.for/main, /proj1.for/main, purp)),
        (SI ↦ (array of reals, /proj1.for/main, /proj1.for/main, purp)),
        (W ↦ (array of reals, /proj1.for/main, /proj1.for/main, purp)),
        (WT ↦ (array of reals, /proj1.for/main, /proj1.for/main, purp)),
        (XTILDE ↦ (array of reals, /proj1.for/main, /proj1.for/main, purp)),
        (XTILDET ↦ (array of reals, /proj1.for/main, /proj1.for/main, purp)),
        (EPSILON ↦ (array of reals, /proj1.for/main, /proj1.for/main, purp)),
        (Graph ↦ (array of reals, /proj1.for/main, /proj1.for/main, purp)),
        (PI ↦ (array of reals, /proj1.for/main, /proj1.for/main, purp)),
        (Nameg ↦ (array of characters, /proj1.for/main, /proj1.for/main, purp))
}
```

3.3.2 Design Recovery Reaching the design recovery task, it is time to provide links between the items identified in the previous task. The first adjustment to the *PROJECT_DESIGN* schema is to give a name to the project. The nature library population brings up another problem with naming. It is possible to refer to the library as the project, or to refer to the final system as the project, or to the individual modules as projects. Naming the modules as projects is chosen. The present system is detailed to the extent of outlining the modules needed to make the system fully functional. Initial additions to the system are most beneficial in these areas. In keeping with this notion, it is reasonable to assume that conversion of individual modules will need to be tracked. Therefore, modules will be considered as projects and given names. This name will be used to track the module and its associated design products. Referring to company policies and standards at this point, a decision is made about the necessary collection of design documents. Since there are no standards presently in place for accomplishing the task at hand, design documents will be created as needed.

The first pass through the program divides it into three parts. The first section initializes variables and opens/creates all of the output files. Section two is a loop that does a number of calculations based on the number of samples selected. The final section closes all of the output files. This shows that all of the work is accomplished in the second portion of the program. The comments in the middle of the program depict this section as sequence of matrix operations. These operations appear to be divided into steps of four to eight statements. This is about the best that can be gleaned from examining the code. This is a textual description of the program. It will be saved in an overview. The information gathered thus far is enough to translate the module into the new system.

In this task *kalman_filter* has been added to the set of reengineering projects. The only design product presently available is the textual description of the module.

PROJECT_DESIGN

project_name : {**kalman_filter**}
project_info : {(**kalman_filter** $\mapsto$ **kalman_filter_text_overview**)}

3.3.3 Information Inspection and Redesign Based on the information recovered in the previous task, the plan developed here during the information inspection is as follows.

1. Develop array objects
2. Develop matrix objects
3. Develop matrix operations
4. Develop a shell with variable declarations

5. Incorporate variable initialization within the shell
6. Incorporate the second portion of the module by adding one collection of steps at a time
7. Incorporate file output

This set of steps is sent to the next task in the renovation phase. The redesign task will not be used since the objective of this research is to capture the original functionality of the modules.

3.3.4 Reimplementation The result of applying this task is a new representation of the existing system. The actual application of this task is discussed in detail in the next chapter.

3.4 Summary

The schemas defined in this chapter can be applied to any project that desires to populate a software library. The chapter that follows discusses the use of these schemas in an actual project.

IV. Implementation

4.1 Introduction

As discussed in the previous chapter, a two-step approach was used to solve the problem. Step one used Byrne's model to plan the project and step two was to implement the plan. This chapter discusses the implementation which essentially followed the steps as outlined in the information inspection task of the last chapter. However, there was the need to do other preliminary research. Initial analysis called for a study of APTAS to get an overall view of the system that needed its library populated. Also, a survey of existing code was conducted to select suitable modules for the test. This chapter covers the preliminaries and then details the renovation.

4.2 Preliminaries

A quick look through the available modules showed that FORTRAN was the primary language used. This led to a study of FORTRAN and its data types. Following this study was a look at CIDL and its data structures. The reason for studying CIDL was that this is the language used in APTAS. Also, it was necessary to compare the data types available in the two languages. Following the study of the languages, APTAS was surveyed to learn how it was constructed. Its primary components consisted of a knowledge base and a collection of library routines called primitives. The structure of APTAS is represented in Appendix A. The top level presents a tracking system to be developed. Questions are presented to the user and the answers determine what additional levels are added to the developing structure. Each new level adds new questions and each new answer may add new levels. This process continues until there is sufficient detail for the knowledge base to select between system primitives.

The research started with gathering samples of modules from existing code that needed to be put into the new system and gathering samples of existing primitives. Appendix D presents the module that was chosen for transformation to the new system. It is a FORTRAN implementation of a kalman filter as outlined in (4). This module was chosen because it is used in several locations in the APTAS structure. The TRACK_DATABASE presented in Appendix B is representative of a primitive module in the APTAS system. This particular module was chosen to show the wide variety of information that must be represented and the many files that are used in maintaining the system knowledge base.

After analyzing the samples and the APTAS structure, the approach to transform the FORTRAN module was further divided as follows: implement the filter in CIDL as a stand-alone module; insert the new module into the knowledge base at the top level; and move the module to its proper place in the hierarchy. This approach was chosen for several reasons. The module presently existed as a stand-alone module. So implementing it in this manner first would allow testing and verification of operation apart from the APTAS system. This would ensure the CIDL representation was correct and would enhance understanding of CIDL syntax and semantics. Inserting the module at the top level would allow easier integration testing. Since all of the intermediate modules are not in place, having this module at the top level will make it much easier to ensure it is invoked. Moving the module to its final level will not require additional testing since integration and functionality have been previously checked. It is a matter of locating and replacing the corresponding module name in the hierarchy. After these preliminaries, research continued with the steps outlined in the previous chapter.

4.3 Source Code Analysis

This phase started with an examination of the files that were used. The source code was contained in a single file. However, examination of the code showed that 16 other files were created during execution. Additionally, the file contained one subroutine to generate random numbers and seven subroutines for matrix operations. FORTRAN library routine that were used in the program were Sin, Cos, and Ran. All variables used in the program were single integers, one-dimensional arrays of reals, or two-dimensional arrays of reals. The relationship between the subroutines and the main program was that all information exchange was accomplished by passing variables to the subroutines so they could be modified.

As part of the source code analysis a search of the APTAS library was performed to find routines that could be reused in the new module. The math module provided Sin, Cos, and Random functions. CIDL did not directly implement arrays, however, there was an existing array module that provided one-dimensional array operations and a matrix module that provided two-dimensional array operations. These modules did not provide all of the required functionality, but they served as good models and were modified and reused. A matrixop module was also available, but it only implemented multiplication and transposition. The following list shows all of the functionality the modules were required to provide.

Math	**Array**	**Matrix**	**Matrixop**
sin	assign	assign	add
cos	index	index	subtract
random	print	print	multiply
	initialize	initialize	inverse (limited to 2×2)
	create	create	transpose

4.4 Design Recovery

An important part of this step is to choose a representation that allows easy transition into the new system and representation of all the recovered information. Since the new system language is CIDL it was also chosen to represent the recovered design and facilitate translation into the new system. The design representation of the existing system was not very complex. Its structure can be described as follows.

```
declare variables
open files
initialize variables
loop
        do filter calculations
        update variables
        write values to files
end loop
close files
```

The variable initializations were sequences of FORTRAN statements. The filter calculations were sequences of statements along with calls to subroutines and library functions. A suitable CIDL structure to represent the program turned out to be the let statement. Its structure is given here.

```
let
        declare variables
in
        open files
        initialize variables
        loop
                do filter calculations
                update variables
                write values to files
        end loop
        close files
end let
```

Here also the program would consist of a sequence of statements, procedure calls, and library function calls.

4.5 *Information Inspection*

The plan developed in this phase to do the actual transformation was straight forward. It would be a phased conversion beginning with the declaration of variables in the new implementation. The next phase would be implementation of the initializations. Once these had been carried out, the sequence would continue with implementing statements, then library function calls, procedure calls, and finally file output. Once file output was complete, comparisons between the original functionality and the new implementation would begin. This would mark the end of converting the selected module to CIDL.

The next phase of the conversion, as discussed previously, would be to put the new module in the knowledge base at the top level so the interfacing could be worked out. This too would need to be a phased task. Beginning at the top level, it would be necessary to insert a new menu option to allow testing of a development module. This involved creating the option, setting necessary variables to deactivate the defaults, and activating levels that implemented the new functionality. Also, the structure of the knowledge base makes it essential that the new module be represented in the following files.

global.desc describing the new *type* and its parameters
global.gsdl-t describing the graphical representation of the new type

global.gsdl-l describing the displayable components
global.synth as a template for generating CIDL
global.form representing the user interface

These items would be developed at this point in the transformation. After this interfacing is completed, testing is performed to ensure proper operation of the new module and its interaction with the knowledge base. The last phase would simply require renaming the module level as dictated by the APTAS structure. This would remove the module from the test status and position it as a normal primitive. Following this plan, the actual redesign can begin.

4.6 Redesign

The initial decision sequence did not require use of the redesign task in system renovation. Since the modules were being used in other programs where they were assumed to function correctly, the idea was to duplicate this functionality without redesign. During the actual reimplementation, however, it became necessary to readdress this decision. As discussed in (8), the relationship between this task and the next is iterative. This proved to be the case in this project. Two particular problems are discussed here.

The transcendental functions used from the FORTRAN library were passed degrees for the calculation. Passing these same numbers to the CIDL functions resulted in errors of several orders of magnitude. This was not discovered until the file output was implemented. To correct this error required a change in the basic design to convert these numbers to radians as required by the CIDL library.

The matrix inverse subroutine in the original module was implemented using the *goto* statement available in FORTRAN. Since CIDL did not have a similar statement, this procedure had to be completely redesigned. Since this version of the filter inverts only 2×2 matrices, a limited procedure was easily developed. Extending this procedure to accommodate 3×3 matrices is not very hard. However, extending the matrix inverse operation for a general n×n matrix will require going to a block structured implementation of an algorithm such as the Gauss-Jordan elimination method (14).

4.7 Reimplementation

The reimplementation followed the plan developed in the information inspection task. Variations in the plan were due to discoveries during the implementation. Two of the problems encountered were discussed in the redesign task since they required a new look at the original redesign decision. Other issues that did not have such a large impact are discussed here as the reimplementation is outlined.

Declaring integer and real variables turned out to be straight forward. Declaring and initializing arrays required modifying the available modules as described previously to provide the needed functionality. Once the array modules were in place and tested, declaring and initializing arrays was easily accomplished. The matrix module built upon the array module and led to further modifications. Finally, the matrixop module was developed and tested.

As the statements were being converted, several undeclared variables were found and had to be declared. FORTRAN allows some variables to be declared at use time and this is not allowed in CIDL. Another problem that developed was with the names chosen for the variables in the original program. They turned out to be reserved words in CIDL and had to be renamed. Some of the original variable types had to changed later also since CIDL had much stronger type checking (integer to real). Translation proceeded by taking FORTRAN statements and mapping them to statements in CIDL. There was continuous testing to ensure functionality was retained.

To complete the CIDL implementation of the module, a complementary test had to be developed. The final numbers produced by the CIDL implementation were not close enough to the figures produced by the original program to verify functionality. It was suspected that the differences were due to the random number generator differences. FORTRAN allowed setting the seed of the generator and CIDL did not. A test was developed by replacing the CIDL random number generator with the sequence of numbers produced by the FORTRAN program. Using these numbers, the two programs had identical output. This marked the end of converting the chosen FORTRAN module to a stand-alone module in CIDL. Reimplementation moved to the task of integrating the module into the knowledge base.

4.8 Summary

APTAS presently contains the overall structure for tracking systems, however, Appendix B lists the modules that are presently still required to completely implement the system. Addition of these modules will not require changing the structure of the present system. The method outlined in this and the previous chapter can be used to put additional modules in the knowledge base. Adding additional capability will modify the structure and will require more changes. The biggest decision to be made will be determining the proper location in the existing hierarchy.

V. Conclusion and Recommendations

5.1 Conclusion

Using design recovery as a means to populate a software library is feasible especially in the case where the structure of the library is in place. This allows the focus to be on obtaining modules with specific functionality. Even if the modules cannot be used *as is*, they can provide guidance for developing new modules during the redesign task of the renovation phase.

The decision to skip the redesign step proved unwise. The model showed that the redesign and reimplementation steps were iterative in nature. Once reimplementation started, problems were encountered that required the redesign step.

The original problem statement in Section 1.3 outlined the four areas that needed to be studied in order to accomplish the library population. The internal formats of the APTAS knowledge base are presented in Appendix A. Characterizing the modules that need to be placed into the library must be accomplished for each module. Following the procedures in Sections 3.2.2 and 3.2.3 will capture the module behavior. A sample plan for mapping the recovered design into the new format is presented in Section 3.3.3. This plan must be customized for each new module.

5.2 Recommendations

The recommendations presented here are divided into areas that need to be addressed in r detail. The ordering of the recommendations is arbitrary.

5.2.1 The Knowledge Base The files that make up the knowledge base have grammars associated with each of them. They are also common to all users of the system. Presently there are no tools specifically designed to maintain these files. Corruption of these files will lead to system wide problems. Two suggestions for helping with this problem are developing a forms based interface for the files and developing a small, stand-alone testbed for new modules. The interface could be made to maintain the grammar for the global.desc, global.form, and other files in the knowledge base limiting the possibility of corrupting these files.

The APTAS Software User's Manual (13) also contains a communication network model as another example of executable specification. It is a much smaller system, but it uses all of the knowledge base component types used in the APTAS system. A similar model would make an excellent testbed for new modules for the APTAS system. This would allow easier testing and integration into the knowledge base.

The items described would aid modifications to the system as it is presently defined. If the structure needs to be changed, additional considerations may be necessary.

5.2.2 Selecting Modules The primitives used in the APTAS system can best be described as communicating sequential processes as defined in (11). They do their processing on information sent by other processes. After the computation, results are sent back to the calling process. This description hints at the best type of modules to select for the knowledge base. Best here refers to structure. Modules that have all of their functionality hidden within a procedure are ideal. All access to the internal structures should be through procedure calls, and the procedures must be able to retain their values across calls. For modules that do not fit this structure, additional work is required to decide which parts of the module are implementing the required behavior and which parts are not needed. The worst type modules would be those that perform interwoven, dissimilar operations.

This idea is also beneficial in the development of candidate modules for other systems. Following the structure outlined above would allow easier reuse of the module in the APTAS system. It sill also promote easier maintenance on the target system.

5.2.3 Translating Modules Due to the large number of files required to complete the system knowledge base, automation of the process seems to be a must. One problem that may face automating the process will be the number of languages used in candidate modules. Each language will require developing a translator. If there are many modules in a given language, this may be beneficial. Since the primitives that are required may have small sizes, it may be easier to code in CIDL using the selected modules as guides than to make translators to convert to CIDL.

Appendix A. *Module Characteristics*

Populating the knowledge base of the APTAS system requires manipulating six files. Five of the files contain a portion of the information characterizing the description of a module. The final file is the actual CIDL implementation of the functionality of the module. This appendix presents the TRACK_DATABASE module as it appears in the five files that describe the module. It also describes the purpose of the file and the information it contains.

A.1 Description

The description of the TRACK_DATABASE is given in the global.desc file. It acts as the help file for the user of the system. It is accessed from the describe_type function of the GDE editor. This file describes the types that are available to the system and lists all the parameters that are contained in an instantiation of a particular type along with their function, and the interface.

```
TRACK_DATABASE:
This module is responsible for the storage,
management and retrieval of all track data.
PARAMETERS TO SPECIFY:
    PLATFORM : platformtype
        (ground or airborne)
    TRACK_BUFFER_SIZE : int
        (range : 1 to 50)
    TRACK_HISTORY_SIZE : int
        (range : 2 to 20)
    PLATFORM_POS_BUFFER_SIZE : int
        (range : 2 to 20)
    MISSION_BUFFER_SIZE : int
        (range : 1 to 10)
    REQUIRED_APPLICATION_MEMORY : int
        (This parameter value is
         calculated during synthesis.
         A value given to it in the
         graphical editor will be
         overwritten by the synthesis
         engine.  It's visibility in
         the graphical editor is merely
         to provide information after
         synthesis.)
    RAM_MEMORY : int
        (how much is available for
         the track database)
@
.:
```

A.2 ICON Representation

This portion of the TRACK_DATABASE is taken from the global.gsdl-t file. It describes how objects are to be represented graphically and how they should be positioned in the GDE editor. Also shown is the information for a relation.

```
TABLE TRACKER_DOMAIN

MODULE : TYPE
    TRACK_DATABASE -> ICON = MED_BLUE_RECT
                           LABEL = TOP BOTH
                           DEFAULT_POSITION = CENTER
                          CONTENTS =
                                  MODULE -> ICON = SM_RECT
                                  -- LABEL = CENTER NAME
                                  DEFAULT_POSITION = CENTER
                          IN_PORT -> ICON = MINI_CIRCLE
                                  LABEL = LEFT NAME
                                  DEFAULT_POSITION = LEFT
                          OUT_PORT -> ICON = MINI_CIRCLE
                                  LABEL = RIGHT NAME
                                  DEFAULT_POSITION = RIGHT
                          DISPLAY RELATIONS;

ASYNC : RELATION
        WIDTH = 0
        COLOR = "Magenta"
        FROM_END = PLAIN
        TO_END = ARROW
        VALID_PAIRS = (MODULE,MODULE)
        EXTRA_ARG;

END
```

A.3 Components

The global.gsdl-1 file describes displayable components of a particular module. It is accessed by the GDE editor.

```
TRACKER_DOMAIN

TRACK_DATABASE : MODULE =
DECLARE
     PLATFORM : PLATFORMTYPE;
     TRACK_BUFFER_SIZE : INT;
     TRACK_HISTORY_SIZE : INT;
     PLATFORM_POS_BUFFER_SIZE : INT;
     MISSION_BUFFER_SIZE : INT;
     REQUIRED_APPLICATION_MEMORY : INT;
     RAM_MEMORY : INT
END
```

A.4 Synthesis

In the global.synth file all of the modules are represented as CIDL templates identifying parameters of the module type. This information is used by the synthesis engine. to generate CIDL.

```
(defvar *prim-modules* nil)
(setq *prim-modules*
  '( (-CDL ((-PA (platform)
                (typeunion ((typeenumlit $ground)
                            (typeenumlit $airborne)))
                (expvoid))
            (-PA (track_buffer_size)          int      (expvoid))
            (-PA (track_history_size)         int      (expvoid))
            (-PA (platform_pos_buffer_size)   int      (expvoid))
            (-PA (mission_buffer_size)        int      (expvoid))
            (-PA (required_application_memory) int     (expvoid))
            (-PA (ram_memory)                 int      (expvoid))
            )
            ()
        (DECLPARAM TRACK_DATABASE_CREATE
            ()
            TRACK_DATABASE (EXPSTRUCT NIL)))

))
```

A.5 Selection

The most important file to be modified is global.form. It contains form information that is presented to the system user and module selection criteria. The form contains the questions that guide the user through the refinement of the system. Based on user input, different combinations of modules are combined. Updating this file requires locating a position in the hierarchy for the module and developing the selection criteria.

```
LEVELS

/TRACKING
"Tracker Boundary Conditions" TRUE

        NOT_DEFAULT EQ NONSENSE | NOT_DEFAULT NE "true" "Default Tracker?"
        STACK
                "true"  VARIABLE_SET(DEFAULT_TRACKER, "true")
                        ACTIVATE_LEVEL(/TRACKING/DEFAULT_TRACKER)
                        ACTIVATE_LEVEL(/TRACKING/DEFAULT_TRACKER/TRACK_DATABASE)
                        ACTIVATE_LEVEL (/TRACKING/DEFAULT_TRACKER/SCAN_TO_TRACK_CORRELATION)
                        ACTIVATE_LEVEL (/TRACKING/DEFAULT_TRACKER/PRESENTATION_PROCESS)
                "false" VARIABLE_SET(DEFAULT_TRACKER, "false")
                        DEACTIVATE_LEVEL(/TRACKING/DEFAULT_TRACKER)
                        DEACTIVATE_LEVEL(/TRACKING/DEFAULT_TRACKER/TRACK_DATABASE)
                        DEACTIVATE_LEVEL (/TRACKING/DEFAULT_TRACKER/SCAN_TO_TRACK_CORRELATION)
                        DEACTIVATE_LEVEL (/TRACKING/DEFAULT_TRACKER/PRESENTATION_PROCESS)
        0;

        "Test Iterations Count"
        NUMERIC [1, 100]
                [1, 100] SAVE_VALUE(TEST_ITERATIONS)
        30;


/TRACKING/DEFAULT_TRACKER
"Default Tracker Top-Level" FALSE
        "No questions, this is a default module."
        CHECKLIST ""
END;

/TRACKING/DEFAULT_TRACKER/TRACK_DATABASE
"Default Tracker Database" FALSE
        "No questions, this is a default module."
        CHECKLIST ""
END
```

```
MODULES

 "TRACKER_ENVIRONMENT"
      "Sensor_Data"                "SENSOR_MODEL"
         {"ITERATIONS" = TEST_ITERATIONS;
          "PERTURBATION_FACTOR" = 0.02;
          "TARGET_SPECS" = ~"[[COMPUTE_FUN = ~DEFAULT_EQ1 ;
                             ARGS = [0.089; 0.009; 1.78;
                                     1.16; 92.31; 8.23]];
                            [COMPUTE_FUN = ~DEFAULT_EQ2 ;
                             ARGS = [0.191; 0.083; 2.46;
                                     2.09; 81.01; -7.23]]]";
          "SENSOR" = ~"~SENSOR_TYPE"}
   default_tracker EQ "true"
      "Tracker"                 "TARGET_TRACKER"
   default_tracker NE "true"
      "Tracker"                 "NEW_TRACKER"

      "Output"                  "OUTPUT_DISPLAY"
      {"TABLE_DATA_FILE" =
         "data_files/default_tracker_table_data.txt";
       "MAP_DATA_FILE" =
         "data_files/default_tracker_map_data.txt";
       "ITERATIONS" = TEST_ITERATIONS}

   REL("Sensor_Scan_Frame_To_Tracker", "Async",
       "Sensor_Data.scan_frame_out", "Tracker.scan_frame_in",
       ~"GenericScanFrame")
   REL("Operator_Query_to_Tracker", "Async",
      "Output.query_out", "Tracker.user_query_in",
      ~"GenericQuery")
   REL("Tracker_Data_to_Display", "Async",
      "Tracker.display_data_out", "Output.reply_in",
       ~"GenericTrackData")
 ;
```

```
"TARGET_TRACKER"
      "TDB"                          "TRACK_DATABASE"
         {"RAM_MEMORY" = RAM_MEMORY;
          "REQUIRED_APPLICATION_MEMORY" = 0;
          "MISSION_BUFFER_SIZE" = 5;
          "PLATFORM_POS_BUFFER_SIZE" = 8;
          "TRACK_HISTORY_SIZE" = 10;
          "TRACK_BUFFER_SIZE" = NUM_OF_TARGETS;
          "PLATFORM" = ~"~PLATFORM_TYPE"}

      "STC"                          "SCAN_TO_TRACK_CORRELATION"
         {"TERMINATED_TENT_TRACK_SAVE_LIMIT" = 5;
          "TERMINATED_REG_TRACK_SAVE_LIMIT" = 20;
          "TENTATIVE_TO_REGULAR_THRESHOLD" = 4;
          "TERMINATION_THRESHOLDS" = ~"[4; 2]";
          "INITIAL_GATE_DELTAS" = ~"[5.0; 5.0]";
          "REDUCED_DELTA_FACTORS" = ~"[0.8; 0.8]";
          "INCREASED_DELTA_FACTORS" = ~"[2.0; 2.0]";
          "DECOYS" = ~"FALSE";
          "FALSE_ALARM_PROBABILITY" = 0.10;
          "TARGET_TRAJECTORY" = ~"MANEUVERING";
          "TARGET_DENSITY" = ~"MEDIUM";
          "TARGET_PROBABILITY_OF_DETECTION" = 0.9;
          "PLATFORM" = ~"~PLATFORM_TYPE";
          "DATABASE" = ~"TDB"}

      "PP"                          "PRESENTATION_PROCESS"
         {"DATABASE" = ~"TDB"}

  DECL("Scan_Frame_In", "In_Port", "")
  DECL("User_Query_In", "In_Port", "")
  DECL("Display_Data_Out", "Out_Port", "")

  REL("STC_Database_TDB_Parameter_Module", "Parameter_Module",
      "STC.database", "TDB", "")
  REL("PP_Database_TDB_Parameter_Module", "Parameter_Module",
      "PP.database", "TDB", "")
  REL("Process_Scan_Frame_In", "Apply_Function",
      "Scan_Frame_In", "STC.Process_Scan_Frame", "GenericScanFrame")
  REL("Process_User_Query_In", "Apply_Function",
      "User_Query_In", "PP.Get_Data_For_Display", "GenericQuery")
  REL("Forward_Display_Data", "Forward_Function_Result",
      "PP.Get_Data_For_Display", "Display_Data_Out",
      "GenericTrackData")
;
```

Appendix B. *System Structure and Population*

This appendix shows the structure of the global.form file as it represents the structure of the system. Structure is represented similar to a file system directory. /TRACKING is the top level of the structure. Anything represented as /TRACKING/SOMENAME would be at the second level of the structure and /TRACKING/SOMENAME/OTHERNAME is at the third level. The LEVELS section of this file represents the hierarchy of enabling forms, and the MODULES section captures the parameters and values. At the bottom of the MODULES section are the lowest level primitives. The primitives without parameters, marked by ';', have not presently been implemented.

```
LEVELS
/TRACKING
     /TRACKING/ZERO_SCAN_SIGNAL_PROCESSING_INTERACTION
     /TRACKING/N_SCAN_REED_SIGNAL_PROCESSING_INTERACTION
     /TRACKING/N_SCAN_TEMPLATE_SIGNAL_PROCESSING_INTERACTION
     /TRACKING/MICROWAVE_RADAR_SENSOR
     /TRACKING/X_BAND_MED_PRF_RADAR_SENSOR
     /TRACKING/PLATFORM
     /TRACKING/PROCESSOR
     /TRACKING/DEFAULT_TRACKER
          /TRACKING/DEFAULT_TRACKER/TRACK_DATABASE
          /TRACKING/DEFAULT_TRACKER/SCAN_TO_TRACK_CORRELATION
          /TRACKING/DEFAULT_TRACKER/PRESENTATION_PROCESS
     /TRACKING/CLUTTER
          /TRACKING/CLUTTER/EST_PRED
          /TRACKING/CLUTTER/EST_PRED/DYN_MOD/VEL_PLUS_ACC
          /TRACKING/CLUTTER/ASSOC
          /TRACKING/CLUTTER/ASSOC/HYP_SELECT_0_SCAN_UNCOORD
          /TRACKING/CLUTTER/ASSOC/HYP_SELECT_0_SCAN_COORD
          /TRACKING/CLUTTER/ASSOC/HYP_SELECT_N_SCAN_UNCOORD
          /TRACKING/CLUTTER/ASSOC/HYP_SELECT_N_SCAN_COORD
          /TRACKING/CLUTTER/PRO_DEM
     /TRACKING/SENSOR
          /TRACKING/SENSOR/NOISE
          /TRACKING/SENSOR/NOISE/DC_CLUTTER
          /TRACKING/SENSOR/NOISE/PC_CLUTTER
          /TRACKING/SENSOR/NOISE/DA_CLUTTER
          /TRACKING/SENSOR/NOISE/PA_CLUTTER
          /TRACKING/SENSOR/NOISE/SEA
          /TRACKING/SENSOR/NOISE/JAMMING
     /TRACKING/SINGLE_CSO
          /TRACKING/SINGLE_CSO/TARGET_CHARS
          /TRACKING/SINGLE_CSO/EST_PRED
          /TRACKING/SINGLE_CSO/EST_PRED/DYN_MOD/VEL_PLUS_ACC
          /TRACKING/SINGLE_CSO/ASSOC
          /TRACKING/SINGLE_CSO/ASSOC/HYP_SELECT_0_SCAN_UNCOORD
          /TRACKING/SINGLE_CSO/ASSOC/HYP_SELECT_0_SCAN_COORD
     /TRACKING/SINGLE_CSO/ASSOC/HYP_SELECT_N_SCAN_UNCOORD
     /TRACKING/SINGLE_CSO/ASSOC/HYP_SELECT_N_SCAN_COORD
     /TRACKING/SINGLE_CSO/PRO_DEM
/TRACKING/GROUPS
     /TRACKING/GROUPS/EST_PRED
     /TRACKING/GROUPS/EST_PRED/DYN_MOD/VEL_PLUS_ACC
     /TRACKING/GROUPS/ASSOC
     /TRACKING/GROUPS/ASSOC/HYP_SELECT_0_SCAN_UNCOORD
     /TRACKING/GROUPS/ASSOC/HYP_SELECT_0_SCAN_COORD
     /TRACKING/GROUPS/ASSOC/HYP_SELECT_N_SCAN_UNCOORD
     /TRACKING/GROUPS/ASSOC/HYP_SELECT_N_SCAN_COORD
     /TRACKING/GROUPS/PRO_DEM
/TRACKING/FORMATIONS
     /TRACKING/FORMATIONS/EST_PRED
     /TRACKING/FORMATIONS/EST_PRED/DYN_MOD/VEL_PLUS_ACC
     /TRACKING/FORMATIONS/ASSOC
     /TRACKING/FORMATIONS/PRO_DEM


MODULES

  "TRACKER_ENVIRONMENT"
     "Sensor_Data"
          "SENSOR_MODEL"
     "Tracker"
          "TARGET_TRACKER"
     "Tracker"
          "NEW_TRACKER"
     "Output"
          "OUTPUT_DISPLAY"

  "TARGET_TRACKER"
     "TDB"
          "TRACK_DATABASE"
     "STC"
          "SCAN_TO_TRACK_CORRELATION"
     "PP"
          "PRESENTATION_PROCESS"

  "NEW_TRACKER"
     "Signal_Processing"
          "ZERO_SCAN_SIG_PROC"
          "N_SCAN_REED_SIG_PROC"
          "N_SCAN_TEMPLATE_MATCHING_SIG_PROC"
     "Process_Single_Objects"
          "SINGLE_OBJECTS"
     "Process_Single_Objects_and_CSOs"
          "SINGLE_OBJECTS"
     "Process_Groups"
          "GROUPS"
     "Process_Formations"
          "FORMATIONS"
     "Process_Clutter"
            "CLUTTER_DISCRETES"

  "SINGLE_OBJECTS"
```

```
    "Estimation_Prediction"
         "SINGLE_ESTIMATION_PREDICTION"
    "Association"
         "SINGLE_ASSOCIATION"
    "Promotion_Demotion"
         "SINGLE_PROMOTION_DEMOTION"

"SINGLE_ESTIMATION_PREDICTION"
    "Dynamical_Model"
         "CONSTANT_VELOCITY_DYNAMICAL_MODEL"
         "CONSTANT_ACCELERATION_DYNAMICAL_MODEL"
         "SINGLE_VEL_PLUS_ACC_DYNAMICAL_MODEL"
    "Measurement_Model"
         "RAE_MEAS_MODEL"
         "VAE_MEAS_MODEL"
         "RVAE_MEAS_MODEL"
    "Plant_Noise_Model"
         "FIXED_PLANT_NOISE_MODEL"
         "KNOWN_AM1_PLANT_NOISE_MODEL"
    "Filter"
         "SIMPLIFIED_GAINS_FILTER"
         "LEAST_SQUARES_FILTER"
         "KALMAN_FILTER"
    "Initial_State_Model"
         "SINGLE_OBS_DERIVED_INIT_STATE_MODEL"
         "SINGLE_GRP_TRK_DERIVED_INIT_STATE_MODEL"

"SINGLE_VEL_PLUS_ACC_DYNAMICAL_MODEL"
    "Model_Generation"
         "GATE_BASED_MODEL_GENERATION"
         "CHI_SQUARE_BASED_MODEL_GENERATION"
         "LIKELIHOOD_BASED_MODEL_GENERATION"
         "PROBABILITY_BASED_MODEL_GENERATION"
    "Model_Score"
         "HIT_MISS_PATTERN_SCORING"
         "CHI_SQUARE_SCORING"
         "LIKELIHOOD_SCORING"
         "PROBABILITY_SCORING"
    "Model_Selection"
         "ZERO_SCAN_DYNAMICAL_MODEL_SELECTION"
         "N_SCAN_DYNAMICAL_MODEL_SELECTION"
    "Model_Transition"
         "VEL_TO_ACC_DYNAMICAL_MODEL_TRANSITION"
         "ACC_TO_VEL_DYNAMICAL_MODEL_TRANSITION"

"SINGLE_ASSOCIATION"
    "Gate_Calculation"
         "RECTANGULAR_GATE_CALCULATION"
         "ELLIPTICAL_GATE_CALCULATION"
         "PARALLELOGRAM_GATE_CALCULATION"
    "Candidate_Generation"
         "GATE_BASED_CANDIDATE_GENERATION"
          "CHI_SQUARE_BASED_CANDIDATE_GENERATION"
          "LIKELIHOOD_BASED_CANDIDATE_GENERATION"
          "PROBABILITY_BASED_CANDIDATE_GENERATION"
```

```
   "Candidate_Scoring"
        "HIT_MISS_PATTERN_SCORING"
        "CHI_SQUARE_SCORING"
        "LIKELIHOOD_SCORING"
        "PROBABILITY_SCORING"
   "Candidate_Selection"
        "Z_SCAN_UNCOORD_HARD_SELECTION"
        "PDA_HYP_SELECTION"
        "GREEDY_HYP_SELECTION"
        "MUNKRES_HYP_SELECTION"
        "MUNKRES_W_CLEANUP_HYP_SELECTION"
        "CLEANUP_HYP_SELECTION"
        "AUCTION_HYP_SELECTION"
        "NETWORK_FLOW_HYP_SELECTION"
        "INT_PROG_HYP_SELECTION"
        "JPDA_HYP_SELECTION"
        "SPLITTING_HYP_SELECTION"
        "SPLITTING_W_MERGING_HYP_SELECTION"
        "SINGLE_N_SCAN_COORDINATED_SELECTION"
        "SINGLE_N_SCAN_COORDINATED_SELECTION"

"SINGLE_N_SCAN_COORDINATED_SELECTION"
   "Hypothesis Generation"
        "K_BEST_HYP_GENERATION"
        "ALL_ABOVE_THRESHOLD_HYP_GENERATION"
   "Hypothesis Selection"
        "N_SCAN_COORD_HYP_SELECTION"

"SINGLE_PROMOTION_DEMOTION"
   "Initiation"
        "ON_ALL_INIT_LOGIC"
        "GATE_BASED_INIT_LOGIC"
        "CHI_SQUARE_BASED_INIT_LOGIC"
        "LIKELIHOOD_BASED_INIT_LOGIC"
        "PROBABILITY_BASED_INIT_LOGIC"
   "Track_Scoring"
        "HIT_MISS_PATTERN_SCORING"
        "CHI_SQUARE_SCORING"
        "LIKELIHOOD_SCORING"
        "PROBABILITY_SCORING"
   "Promote_Logic"
        "M_OF_N_PROMOTE_LOGIC"
        "CHI_SQUARE_TEST_PROMOTE_LOGIC"
        "LIKELIHOOD_TEST_PROMOTE_LOGIC"
        "PROBABILITY_TEST_PROMOTE_LOGIC"
   "Demote_Logic"
        "K_MISS_DEMOTE_LOGIC"
        "CHI_SQUARE_TEST_DEMOTE_LOGIC"
        "LIKELIHOOD_TEST_DEMOTE_LOGIC"
        "PROBABILITY_TEST_DEMOTE_LOGIC"

"GROUPS"
   "Estimation_Prediction"
        "GROUPS_ESTIMATION_PREDICTION"
   "Association"
```

```
        "GROUPS_ASSOCIATION"
   "Promotion_Demotion"
        "GROUPS_PROMOTION_DEMOTION"

"GROUPS_ESTIMATION_PREDICTION"
   "Centroid_Dynamical_Model"
        "CONSTANT_VELOCITY_DYNAMICAL_MODEL"
        "CONSTANT_ACCELERATION_DYNAMICAL_MODEL"
        "GROUPS_VEL_PLUS_ACC_DYNAMICAL_MODEL"
   "Measurement_Model"
        "RAE_MEAS_MODEL"
        "VAE_MEAS_MODEL"
        "RVAE_MEAS_MODEL"
   "Plant_Noise_Model"
        "FIXED_PLANT_NOISE_MODEL"
        "KNOWN_AMI_PLANT_NOISE_MODEL"
   "Filter"
        "SIMPLIFIED_GAINS_FILTER"
        "LEAST_SQUARES_FILTER"
        "KALMAN_FILTER"
   "Initial_State_Model"
        "GROUPS_INIT_STATE_MODEL"

"GROUPS_VEL_PLUS_ACC_DYNAMICAL_MODEL"
   "Model_Generation"
        "GATE_BASED_MODEL_GENERATION"
        "CHI_SQUARE_BASED_MODEL_GENERATION"
        "LIKELIHOOD_BASED_MODEL_GENERATION"
        "PROBABILITY_BASED_MODEL_GENERATION"
   "Model_Score"
        "HIT_MISS_PATTERN_SCORING"
        "CHI_SQUARE_SCORING"
        "LIKELIHOOD_SCORING"
        "PROBABILITY_SCORING"
   "Model_Selection"
        "ZERO_SCAN_DYNAMICAL_MODEL_SELECTION"
            "N_SCAN_DYNAMICAL_MODEL_SELECTION"
   "Model_Transition"
        "VEL_TO_ACC_DYNAMICAL_MODEL_TRANSITION"
        "ACC_TO_VEL_DYNAMICAL_MODEL_TRANSITION"

"GROUPS_ASSOCIATION"
   "Gate_Calculation"
        "RECTANGULAR_GATE_CALCULATION"
        "ELLIPTICAL_GATE_CALCULATION"
        "PARALLELOGRAM_GATE_CALCULATION"
   "Candidate_Generation"
        "GROUPS_INDEPENDENT_HYP_GENERATION"
        "GROUPS_DEPENDENT_HYP_GENERATION"
   "Candidate_Scoring"
        "HIT_MISS_PATTERN_SCORING"
        "CHI_SQUARE_SCORING"
        "LIKELIHOOD_SCORING"
        "PROBABILITY_SCORING"
   "Candidate_Selection"
```

```
          "Z_SCAN_UNCOORD_HARD_SELECTION"
          "PDA_HYP_SELECTION"
          "GREEDY_HYP_SELECTION"
          "MUNKRES_HYP_SELECTION"
          "MUNKRES_W_CLEANUP_HYP_SELECTION"
          "CLEANUP_HYP_SELECTION"
          "AUCTION_HYP_SELECTION"
          "NETWORK_FLOW_HYP_SELECTION"
          "INT_PROG_HYP_SELECTION"
          "JPDA_HYP_SELECTION"
          "SPLITTING_HYP_SELECTION"
          "SPLITTING_W_MERGING_HYP_SELECTION"
          "GROUPS_N_SCAN_COORDINATED_SELECTION"
          "GROUPS_N_SCAN_COORDINATED_SELECTION"

"GROUPS_N_SCAN_COORDINATED_SELECTION"
   "Hypothesis Generation"
          "K_BEST_HYP_GENERATION"
          "ALL_ABOVE_THRESHOLD_HYP_GENERATION"
   "Hypothesis Selection"
          "N_SCAN_COORD_HYP_SELECTION"

"GROUPS_PROMOTION_DEMOTION"
   "Initiation"
          "ON_ALL_INIT_LOGIC"
          "GATE_BASED_INIT_LOGIC"
          "CHI_SQUARE_BASED_INIT_LOGIC"
          "LIKELIHOOD_BASED_INIT_LOGIC"
          "PROBABILITY_BASED_INIT_LOGIC"
   "Track_Scoring"
          "HIT_MISS_PATTERN_SCORING"
          "CHI_SQUARE_SCORING"
          "LIKELIHOOD_SCORING"
          "PROBABILITY_SCORING"
   "Promote_Logic"
          "M_OF_N_PROMOTE_LOGIC"
          "CHI_SQUARE_TEST_PROMOTE_LOGIC"
          "LIKELIHOOD_TEST_PROMOTE_LOGIC"
          "PROBABILITY_TEST_PROMOTE_LOGIC"
   "Demote_Logic"
          "K_MISS_DEMOTE_LOGIC"
          "CHI_SQUARE_TEST_DEMOTE_LOGIC"
          "LIKELIHOOD_TEST_DEMOTE_LOGIC"
          "PROBABILITY_TEST_DEMOTE_LOGIC"

"FORMATIONS"
   "Estimation_Prediction"
          "FORMATIONS_ESTIMATION_PREDICTION"
   "Association"
          "FORMATIONS_ASSOCIATION"
   "Promotion_Demotion"
          "FORMATIONS_PROMOTION_DEMOTION"

"FORMATIONS_ESTIMATION_PREDICTION"
   "Dynamical_Model"
```

```
        "CONSTANT_VELOCITY_DYNAMICAL_MODEL"
        "CONSTANT_ACCELERATION_DYNAMICAL_MODEL"
        "FORMATIONS_VEL_PLUS_ACC_DYNAMICAL_MODEL"
   "Measurement_Model"
        "RAE_MEAS_MODEL"
        "VAE_MEAS_MODEL"
        "RVAE_MEAS_MODEL"
   "Plant_Noise_Model"
        "FIXED_PLANT_NOISE_MODEL"
        "KNOWN_AM1_PLANT_NOISE_MODEL"
   "Filter"
        "SIMPLIFIED_GAINS_FILTER"
        "LEAST_SQUARES_FILTER"
        "KALMAN_FILTER"
   "Initial_State_Model"
        "FORMATIONS_INIT_STATE_MODEL"

"FORMATIONS_VEL_PLUS_ACC_DYNAMICAL_MODEL"
   "Model_Generation"
        "GATE_BASED_MODEL_GENERATION"
        "CHI_SQUARE_BASED_MODEL_GENERATION"
        "LIKELIHOOD_BASED_MODEL_GENERATION"
        "PROBABILITY_BASED_MODEL_GENERATION"
   "Model_Score"
        "HIT_MISS_PATTERN_SCORING"
        "CHI_SQUARE_SCORING"
        "LIKELIHOOD_SCORING"
        "PROBABILITY_SCORING"
   "Model_Selection"
        "ZERO_SCAN_DYNAMICAL_MODEL_SELECTION"
        "N_SCAN_DYNAMICAL_MODEL_SELECTION"
   "Model_Transition"
        "VEL_TO_ACC_DYNAMICAL_MODEL_TRANSITION"
        "ACC_TO_VEL_DYNAMICAL_MODEL_TRANSITION"

"FORMATIONS_ASSOCIATION"
   "Gate_Calculation"
        "RECTANGULAR_GATE_CALCULATION"
        "ELLIPTICAL_GATE_CALCULATION"
        "PARALLELOGRAM_GATE_CALCULATION"
   "Correlation"
        "FORMATIONS_TEMPLATE_CORR"
        "FORMATIONS_STATISTICAL_CORR"
        "FORMATIONS_INTERTRACK_CORR"

"FORMATIONS_PROMOTION_DEMOTION"
    "Initiation"
        "ON_ALL_INIT_LOGIC"
        "GATE_BASED_INIT_LOGIC"
        "CHI_SQUARE_BASED_INIT_LOGIC"
        "LIKELIHOOD_BASED_INIT_LOGIC"
        "PROBABILITY_BASED_INIT_LOGIC"
   "Track_Scoring"
        "HIT_MISS_PATTERN_SCORING"
        "CHI_SQUARE_SCORING"
        "LIKELIHOOD_SCORING"
        "PROBABILITY_SCORING"
    "Promote_Logic"
        "M_OF_N_PROMOTE_LOGIC"
        "CHI_SQUARE_TEST_PROMOTE_LOGIC"
        "LIKELIHOOD_TEST_PROMOTE_LOGIC"
        "PROBABILITY_TEST_PROMOTE_LOGIC"
    "Demote_Logic"
        "K_MISS_DEMOTE_LOGIC"
        "CHI_SQUARE_TEST_DEMOTE_LOGIC"
        "LIKELIHOOD_TEST_DEMOTE_LOGIC"
        "PROBABILITY_TEST_DEMOTE_LOGIC"

"CLUTTER_DISCRETES"
    "Estimation_Prediction"
        "CLUTTER_ESTIMATION_PREDICTION"
    "Association"
        "CLUTTER_ASSOCIATION"
    "Promotion_Demotion"
        "CLUTTER_PROMOTION_DEMOTION"

"CLUTTER_ESTIMATION_PREDICTION"
    "Dynamical_Model"
        "CONSTANT_VELOCITY_DYNAMICAL_MODEL"
        "CONSTANT_ACCELERATION_DYNAMICAL_MODEL"
        "CLUTTER_VEL_PLUS_ACC_DYNAMICAL_MODEL"
    "Measurement_Model"
        "RAE_MEAS_MODEL"
        "VAE_MEAS_MODEL"
        "RVAE_MEAS_MODEL"
    "Plant_Noise_Model"
        "FIXED_PLANT_NOISE_MODEL"
        "KNOWN_AM1_PLANT_NOISE_MODEL"
    "Filter"
        "SIMPLIFIED_GAINS_FILTER"
        "LEAST_SQUARES_FILTER"
        "KALMAN_FILTER"
    "Initial_State_Model"
        "CLUTTER_INIT_STATE_MODEL"

"CLUTTER_VEL_PLUS_ACC_DYNAMICAL_MODEL"
    "Model_Generation"
        "GATE_BASED_MODEL_GENERATION"
        "CHI_SQUARE_BASED_MODEL_GENERATION"
        "LIKELIHOOD_BASED_MODEL_GENERATION"
        "PROBABILITY_BASED_MODEL_GENERATION"
    "Model_Score"
        "HIT_MISS_PATTERN_SCORING"
        "CHI_SQUARE_SCORING"
        "LIKELIHOOD_SCORING"
        "PROBABILITY_SCORING"
    "Model_Selection"
        "ZERO_SCAN_DYNAMICAL_MODEL_SELECTION"
        "N_SCAN_DYNAMICAL_MODEL_SELECTION"
    "Model_Transition"
```

```
        "VEL_TO_ACC_DYNAMICAL_MODEL_TRANSITION"
        "ACC_TO_VEL_DYNAMICAL_MODEL_TRANSITION"

"CLUTTER_ASSOCIATION"
   "Gate_Calculation"
        "RECTANGULAR_GATE_CALCULATION"
        "ELLIPTICAL_GATE_CALCULATION"
        "PARALLELOGRAM_GATE_CALCULATION"
   "Candidate_Generation"
        "GATE_BASED_CANDIDATE_GENERATION"
         "CHI_SQUARE_BASED_CANDIDATE_GENERATION"
         "LIKELIHOOD_BASED_CANDIDATE_GENERATION"
         "PROBABILITY_BASED_CANDIDATE_GENERATION"
   "Candidate_Scoring"
        "HIT_MISS_PATTERN_SCORING"
        "CHI_SQUARE_SCORING"
        "LIKELIHOOD_SCORING"
        "PROBABILITY_SCORING"
   "Candidate_Selection"
        "Z_SCAN_UNCOORD_HARD_SELECTION"
        "PDA_HYP_SELECTION"
        "GREEDY_HYP_SELECTION"
        "MUNKRES_HYP_SELECTION"
        "MUNKRES_W_CLEANUP_HYP_SELECTION"
        "CLEANUP_HYP_SELECTION"
        "AUCTION_HYP_SELECTION"
        "NETWORK_FLOW_HYP_SELECTION"
        "INT_PROG_HYP_SELECTION"
        "JPDA_HYP_SELECTION"
        "SPLITTING_HYP_SELECTION"
        "SPLITTING_W_MERGING_HYP_SELECTION"
        "CLUTTER_N_SCAN_COORDINATED_SELECTION"
        "CLUTTER_N_SCAN_COORDINATED_SELECTION"

"CLUTTER_N_SCAN_COORDINATED_SELECTION"
   "Hypothesis Generation"
        "K_BEST_HYP_GENERATION"
        "ALL_ABOVE_THRESHOLD_HYP_GENERATION"
   "Hypothesis Selection"
        "N_SCAN_COORD_HYP_SELECTION"

"CLUTTER_PROMOTION_DEMOTION"
        "Initiation"
        "ON_ALL_INIT_LOGIC"
        "GATE_BASED_INIT_LOGIC"
        "CHI_SQUARE_BASED_INIT_LOGIC"
        "LIKELIHOOD_BASED_INIT_LOGIC"
        "PROBABILITY_BASED_INIT_LOGIC"
   "Track_Scoring"
        "HIT_MISS_PATTERN_SCORING"
        "CHI_SQUARE_SCORING"
        "LIKELIHOOD_SCORING"
        "PROBABILITY_SCORING"
   "Promote_Logic"
         [illegible]_OF_N_PROMOTE_LOGIC"
```

```
          "CHI_SQUARE_TEST_PROMOTE_LOGIC"
          "LIKELIHOOD_TEST_PROMOTE_LOGIC"
          "PROBABILITY_TEST_PROMOTE_LOGIC"
     "Demote_Logic"
          "K_MISS_DEMOTE_LOGIC"
          "CHI_SQUARE_TEST_DEMOTE_LOGIC"
          "LIKELIHOOD_TEST_DEMOTE_LOGIC"
          "PROBABILITY_TEST_DEMOTE_LOGIC"

"ZERO_SCAN_SIG_PROC"                           ;
"N_SCAN_REED_SIG_PROC"                         ;
"N_SCAN_TEMPLATE_MATCHING_SIG_PROC"            ;

"GATE_BASED_MODEL_GENERATION"                  ;
"CHI_SQUARE_BASED_MODEL_GENERATION"            ;
"LIKELIHOOD_BASED_MODEL_GENERATION"            ;
"PROBABILITY_BASED_MODEL_GENERATION"           ;

"ZERO_SCAN_DYNAMICAL_MODEL_SELECTION"          ;
"N_SCAN_DYNAMICAL_MODEL_SELECTION"             ;

"VEL_TO_ACC_DYNAMICAL_MODEL_TRANSITION"        ;
"ACC_TO_VEL_DYNAMICAL_MODEL_TRANSITION"        ;

"RAE_MEAS_MODEL"                               ;
"VAE_MEAS_MODEL"                               ;
"RVAE_MEAS_MODEL"                              ;

"FIXED_PLANT_NOISE_MODEL"                      ;
"KNOWN_AM1_PLANT_NOISE_MODEL"                  ;

"SIMPLIFIED_GAINS_FILTER"                      ;
"LEAST_SQUARES_FILTER"                         ;
"KALMAN_FILTER"                                ;

"SINGLE_OBS_DERIVED_INIT_STATE_MODEL"          ;
"SINGLE_GRP_TRK_DERIVED_INIT_STATE_MODEL"      ;

"CONSTANT_VELOCITY_DYNAMICAL_MODEL"            ;
"CONSTANT_ACCELERATION_DYNAMICAL_MODEL"        ;

"K_BEST_HYP_GENERATION"                        ;
"ALL_ABOVE_THRESHOLD_HYP_GENERATION"           ;

"N_SCAN_COORD_HYP_SELECTION"                   ;

"RECTANGULAR_GATE_CALCULATION"                 ;
"ELLIPTICAL_GATE_CALCULATION"                  ;
"PARALLELOGRAM_GATE_CALCULATION"               ;

"GATE_BASED_CANDIDATE_GENERATION"              ;
"CHI_SQUARE_BASED_CANDIDATE_GENERATION"        ;
"LIKELIHOOD_BASED_CANDIDATE_GENERATION"        ;
"PROBABILITY_BASED_CANDIDATE_GENERATION"       ;
```

```
"Z_SCAN_UNCOORD_HARD_SELECTION"          ;
"PDA_HYP_SELECTION"                      ;
"GREEDY_HYP_SELECTION"                   ;
"MUNKRES_HYP_SELECTION"                  ;
"MUNKRES_W_CLEANUP_HYP_SELECTION"        ;
"CLEANUP_HYP_SELECTION"                  ;
"AUCTION_HYP_SELECTION"                  ;
"NETWORK_FLOW_HYP_SELECTION"             ;
"INT_PROG_HYP_SELECTION"                 ;
"JPDA_HYP_SELECTION"                     ;
"SPLITTING_HYP_SELECTION"                ;
"SPLITTING_W_MERGING_HYP_SELECTION"      ;

"ON_ALL_INIT_LOGIC"                      ;
"GATE_BASED_INIT_LOGIC"                  ;
"CHI_SQUARE_BASED_INIT_LOGIC"            ;
"LIKELIHOOD_BASED_INIT_LOGIC"            ;

"HIT_MISS_PATTERN_SCORING"               ;
"CHI_SQUARE_SCORING"                     ;
"LIKELIHOOD_SCORING"                     ;
"PROBABILITY_SCORING"                    ;

"M_OF_N_PROMOTE_LOGIC"                   ;
"CHI_SQUARE_TEST_PROMOTE_LOGIC"          ;
"LIKELIHOOD_TEST_PROMOTE_LOGIC"          ;
"PROBABILITY_TEST_PROMOTE_LOGIC"         ;

"K_MISS_DEMOTE_LOGIC"                    ;
"CHI_SQUARE_TEST_DEMOTE_LOGIC"           ;
"LIKELIHOOD_TEST_DEMOTE_LOGIC"           ;
"PROBABILITY_TEST_DEMOTE_LOGIC"          ;

"GROUPS_INDEPENDENT_HYP_GENERATION"      ;
"GROUPS_DEPENDENT_HYP_GENERATION"        ;

"GROUPS_INIT_STATE_MODEL"                ;
"FORMATIONS_INIT_STATE_MODEL"            ;
"CLUTTER_INIT_STATE_MODEL"               ;

"FORMATIONS_TEMPLATE_CORR"               ;
"FORMATIONS_STATISTICAL_CORR"            ;
"FORMATIONS_INTERTRACK_CORR"             ;
```

Appendix C. *Summary of Z Notation*

LHS $\widehat{=}$ RHS	Definition of LHS as syntactically equivalent to RHS
x: T	Declaration of x as type T
x?: T	Declaration of x as type T used as input to an operation
x!: T	Declaration of x as type T used as output by an operation
x	Value of x before an operation
x'	Value of x after an operation
$\emptyset$	The empty set
$\mathbb{P}$ X	The power set of X
$\mathbb{F}$ X	The finite subset x of X
S $\subset$ T	S is a subset of T
t $\in$ S	t is an element of S
t $\notin$ S	t is not an element of S
S $\cup$ T	Set union
S $\cap$ T	Set intersection
$T_1 \times T_2$	Cartesian product
X $\nrightarrow$ Y	The set of partial functions from X to Y
l $\mapsto$ m	l is related to m
Ξ Schema	Include but do not change the schema
Δ Schema	Include and allow change to the schema
S $\lhd$ T	Domain subtraction
$R_1 \oplus R_2$	Overriding
$\mid$	Such that

Appendix D. *Selected FORTRAN Module*

```
C  HOMEWORK PROJECT #1 - DISCRETE KALMAN FILTER
C                               GREG BIERMAN
C
         PROGRAM PROJ1

C    VARIABLE LIST
         IMPLICIT NONE
         INTEGER K,C,I1,I2,JS,IOPT,NGR,NPT,J
         REAL X(100),Y(100),WX(100),WY(100),XHATN(2,1),XHATOLD(2,1)
         REAL VX(100),VY(100),Z(2,1),XHAT(2,1),ZHAT(2,1),NU(2,1)
         REAL P(2,2),F(2,2),FT(2,2),R(2,2),PN(2,2)
         REAL TEMP1(2,2),TEMP2(2,2),Q(2,2),H(2,2),S(2,2),HT(2,2),SI(2,2)
         REAL W(2,2),WT(2,2),XTILDE(2,1),XTILDET(1,2),EPSILON(1,1)
         REAL Graph(20,200),PI(2,2)
         CHARACTER*16 Nameg(20)
         COMMON /SEED/ I1,I2
         I1=12345
         I2=54321

C    OPEN FILES USED TO STORE DATA
         Open(Unit=8,File='KALMAN.PLT',FORM='unformatted',Status='new')
         Open(Unit=9,File='KALMAN.PL',Status='new')
         Open(Unit=10,File='XTRUTH.PRN',Status='new')
         Open(Unit=11,File='YTRUTH.PRN',Status='new')
         Open(Unit=12,File='XMEAS.PRN',Status='new')
         Open(Unit=13,File='YMEAS.PRN',Status='new')
         Open(Unit=14,File='XEST.PRN',Status='new')
         Open(Unit=15,File='YEST.PRN',Status='new')
         Open(Unit=16,File='XNOISE.PRN',Status='new')
         Open(Unit=17,File='YNOISE.PRN',Status='new')
         Open(Unit=20,File='ZXNOISE.PRN',Status='new')
         Open(Unit=21,File='ZYNOISE.PRN',Status='new')
         Open(Unit=22,File='P11.PRN',Status='new')
         Open(Unit=23,File='P22.PRN',Status='new')
         Open(Unit=24,File='EPSILON.PRN',Status='new')
         Open(Unit=25,File='XTILDE.PRN',Status='new')

C    START MAIN PROGRAM (73 POINTS COLLECTED)
         C = 73

C    COMPUTE MODELLING NOISE (SET TO ZERO)
         DO K=1,C
              CALL NOISE(0,0,WX(K),12)
              CALL NOISE(0,0,WY(K),12)
              Write(16,*) WX(K)
              Write(17,*) WY(K)
         end do

C    SET INITIAL VALUES
         X(1)=10.0
         Y(1)=0.0
```

```
      Z(1,1)=10.0
      Z(2,1)=0.0
      P(1,1) = 1.0
      P(1,2) = 0.0
      P(2,1) = 0.0
      P(2,2) = 1.0
      F(1,1) = COSD(5.0)
      F(1,2) = -0.5*SIND(5.0)
      F(2,1) = 2*SIND(5.0)
      F(2,2) = COSD(5.0)
      CALL MTXTRP (F,FT,2,2)
      Q(1,1) = 0.0
      Q(1,2) = 0.0
      Q(2,1) = 0.0
      Q(2,2) = 0.0
      H(1,1) = 1.0
      H(1,2) = 0.0
      H(2,1) = 0.0
      H(2,2) = 1.0
      CALL MTXTRP (H,HT,2,2)
      R(1,2) = 0.0
      R(2,1) = 0.0
      IS = 0
      XHATOLD(1,1) = 10.0
      XHATOLD(2,1) = 0.0

C   START MAIN LOOP
      DO K=1,C

C   WRITE INITIAL VALUES TO FILES
      Graph(1,K) = K
      Write(10,*) X(K)
      Write(11,*) Y(K)
      Graph(2,K) = X(K)
      Graph(3,K) = Y(K)
      Graph(4,K) = Z(1,1)
      Graph(5,K) = Z(2,1)
      Write(12,*) Z(1,1)
      Write(13,*) Z(2,1)
      Graph(8,K) = P(1,1)
      Graph(9,K) = P(2,2)
      Write(22,*) P(1,1)
      Write(23,*) P(2,2)
      Graph(6,K) = XHATOLD(1,1)
      Graph(7,K) = XHATOLD(2,1)
      Write(14,*) XHATOLD(1,1)
      Write(15,*) XHATOLD(2,1)

C   NORMALIZED STATE ERROR (EPSILON = XTILDET*PINV*XTILDE)
C   XHATOLD = XHATN
      XTILDE(1,1) = X(K)-XHATOLD(1,1)
      XTILDE(2,1) = Y(K)-XHATOLD(2,1)
      CALL MTXTRP (XTILDE,XTILDET,2,1)
      CALL MTXINV(P,PI,TEMP1,2,IS)
      CALL MTXMUL(XTILDET,PI,TEMP1,1,2,2)
```

```
          CALL MTXMUL(TEMP1,XTILDE,EPSILON,1,2,1)
          Graph(10,K) = EPSILON(1,1)
          Write(24,*) EPSILON(1,1)
          Graph(11,K) = XTILDE(1,1)
          Write(25,*) XTILDE(1,1)

C     STATE PREDICTION; XHAT = F*XHATOLD
          XHAT(1,1)=F(1,1)*XHATOLD(1,1)+F(1,2)*XHATOLD(2,1)
          XHAT(2,1)=F(2,1)*XHATOLD(1,1)+F(2,2)*XHATOLD(2,1)

C     TRUTH;  X(K+1) = F*X(K)
          X(K+1) = F(1,1)*X(K)+F(1,2)*Y(K)+WX(K)
          Y(K+1) = F(2,1)*X(K)+F(2,2)*Y(K)+WY(K)

C     COMPUTE MEASUREMENT NOISE
          R(1,1) = 0.2*ABS(X(K+1))
          R(2,2) = 0.2*ABS(Y(K+1))
          CALL NOISE(0,R(1,1),VX(K+1),12)
          CALL NOISE(0,R(2,2),VY(K+1),12)
          Write(20,*) VX(K+1)
          Write(21,*) VY(K+1)
          Graph(12,K+1) = VX(K+1)
          Graph(13,K+1) = VY(K+1)

C     MEASUREMENT; Z = X+V
          Z(1,1)=X(K+1)+VX(K+1)
          Z(2,1)=Y(K+1)+VY(K+1)

C     MEASUREMENT PREDICTION (ZHAT=H*XHAT)
          CALL MTXMUL(H,XHAT,ZHAT,2,2,1)

C     INNOVATION (NU=Z-ZHAT)
          CALL MTXSUB(Z,ZHAT,NU,2,1)

C     STATE PREDICTION COVARIANCE (P=F*P*FT+Q)
          CALL MTXMUL(F,P,TEMP1,2,2,2)
          CALL MTXMUL(TEMP1,FT,TEMP2,2,2,2)
          CALL MTXADD(TEMP2,Q,P,2,2)

C     INNOVATION COVARIANCE (S=H*P*HT+R)
          CALL MTXMUL(H,P,TEMP1,2,2,2)
          CALL MTXMUL(TEMP1,HT,TEMP2,2,2,2)
          CALL MTXADD(TEMP2,R,S,2,2)

C     FILTER GAIN (W=P*HT*SI)
          CALL MTXINV(S,SI,TEMP1,2,IS)
          CALL MTXMUL(P,HT,TEMP1,2,2,2)
          CALL MTXMUL(TEMP1,SI,W,2,2,2)

C     UPDATED STATE COVARIANCE (PN=P-W*S*WT)
          CALL MTXTRP (W,WT,2,2)
          CALL MTXMUL(W,S,TEMP1,2,2,2)
          CALL MTXMUL(TEMP1,WT,TEMP2,2,2,2)
          CALL MTXSUB(P,TEMP2,PN,2,2)
          P(1,1) = PN(1,1)
```

```
      P(1,2) = PN(1,2)
      P(2,1) = PN(2,1)
      P(2,2) = PN(2,2)

C     UPDATED STATE ESTIMATE (XHATN=XHAT+W*NU)
      CALL MTXMUL(W,NU,TEMP1,2,2,1)
      CALL MTXADD(XHAT,TEMP1,XHATN,2,1)
      XHATOLD(1,1) = XHATN(1,1)
      XHATOLD(2,1) = XHATN(2,1)
      end do

C     PLOT ROUTINE FOR OUTPUT DATA
      NGR=13
      Nameg(1) = 'K'
      Nameg(2) = 'X TRUTH'
      Nameg(3) = 'Y TRUTH'
      Nameg(4) = 'X MEASURED'
      Nameg(5) = 'Y MEASURED'
      Nameg(6) = 'X ESTIMATE'
      Nameg(7) = 'Y ESTIMATE'
      Nameg(8) = 'P11, X COVAR'
      Nameg(9) = 'P22, Y COVAR'
      Nameg(10) = 'EPSILON'
      Nameg(11) = 'XTILDE'
      Nameg(12) = 'MEAS NOISE X'
      Nameg(13) = 'MEAS NOISE Y'

      IOPT = 1
      NPT = 73
      WRITE(8) NGR,NPT,IOPT
      WRITE(8) (Nameg(j),j=1,NGR)
      WRITE(8) ((Graph(j,k),k=1,NPT),j=1,NGR)
      WRITE(9,*) NGR,NPT,IOPT
      WRITE(9,*) (Nameg(j),j=1,NGR)
      WRITE(9,*) ((Graph(j,k),k=1,NPT),j=1,NGR)

C     CLOSE DATA FILES
      Close(8)
      Close(9)
      Close(10)
      Close(11)
      Close(12)
      Close(13)
      Close(14)
      Close(15)
      Close(16)
      Close(17)
      Close(18)
      Close(19)
      Close(20)
      Close(21)
      Close(22)
      Close(23)
      Close(24)
      Close(25)
```

```
        End

C
C THIS PROGRAM CALLS RANDOM GENERATOR TO GENERATE GAUSSIAN NOISE.
C
C       N  is set  12
C
        SUBROUTINE NOISE(XMEAN,VARIANCE,RNDMN,N)
        REAL A,Y,RNDMN,XMEAN,VARIANCE
        INTEGER N
        COMMON /SEED/ I1,I2
        A=0.0
        DO 1 I=1,N
        Y=RAN(I1,I2)
        A=A+Y
00001   CONTINUE
        RNDMN=(A-N*0.5)*SQRT(VARIANCE)+XMEAN
        RETURN
        END

C*************** SUBROUTINES OF MATRIX OPERATIONS ***************
C
        SUBROUTINE MTXMUL (A,B,C,N1,N2,N3)
C        A IS N1*N2; B IS N2*N3; C = A*B IS N1*N3
        real A(N1,N2), B(N2,N3), C(N1,N3)
        DO  I = 1,N1
          DO  J = 1,N3
            C(I,J) = 0.0
            DO  K = 1,N2
              C(I,J) = C(I,J)+A(I,K)*B(K,J)
            end do
          end do
        end do
        RETURN
        END

        SUBROUTINE MTXADD (A,B,C,N1,N2)
C        C = A+B; ALL ARE N1*N2
        real A(N1,N2), B(N1,N2), C(N1,N2)
        DO  I = 1,N1
          DO  J = 1,N2
            C(I,J) = A(I,J)+B(I,J)
          end do
        end do
        RETURN
        END

        SUBROUTINE MTXSUB (A,B,C,N1,N2)
C        C = A-B; ALL ARE N1*N2
        real A(N1,N2), B(N1,N2), C(N1,N2)
        DO  I = 1,N1
          DO  J = 1,N2
            C(I,J) = A(I,J)-B(I,J)
          end do
```

```
      end do
      RETURN
      END

      SUBROUTINE MTXZRO (A,N1,N2)
C      A = 0; A IS N1*N2
      real A(N1,N2)
      DO  I = 1,N1
        DO  J = 1,N2
          A(I,J) = 0.0
        end do
      end do
      RETURN
      END

       SUBROUTINE MTXTRP (A,B,N1,N2)
C      B = TRANSPOSE OF A; A IS N1*N2 AND B IS N2*N1
      real A(N1,N2), B(N2,N1)
      DO  J = 1,N2
        DO  I = 1,N1
          B(J,I) = A(I,J)
        end do
      end do
      RETURN
      END

      SUBROUTINE MTXINV(A,AINV,B,KC,IS)
C      AINV=INVERSE OF A, BOTH ARE KC*KC
C      B IS A WORKING ARRAY
C     WHEN IS=0, SUCCESSFUL RETURN. IS=1 A IS SINGULAR.
c
      real A(KC,KC),AINV(KC,KC),B(KC,KC)
      REAL TEMP,COMP,EPSKIP
      N=1
      IS=1
      EPSKIP=1.0E-35
      DO 1 I=1,KC
      DO 1 J=1,KC
      AINV(I,J)=0.0
00001 B(I,J)=A(I,J)
      DO 2 I=1,KC
      AINV(I,I)=1.0
00002 CONTINUE
      DO 3 I=1,KC
      COMP=0.0
      K=I
00006 IF(ABS(B(K,I))-ABS(COMP))5,5,4
00004 COMP=B(K,I)
      N=K
00005 K=K+1
      IF(K-KC)6,6,7
00007 IF(B(N,I))8,51,8
00008 IF(N-I)51,12,9
00009 DO 10 M=1,KC
      TEMP=B(I,M)
      B(I,M)=B(N,M)
      B(N,M)=TEMP
      TEMP=AINV(I,M)
      AINV(I,M)=AINV(N,M)
00010 AINV(N,M)=TEMP
00012 CONTINUE
      TEMP=B(I,I)
      DO 13 M=1,KC
      AINV(I,M)=AINV(I,M)/TEMP
00013 B(I,M)=B(I,M)/TEMP
      DO 16 J=1,KC
      IF(J-I)14,16,14
00014 IF(B(J,I))15,16,15
00015 CONTINUE
      TEMP=B(J,I)
      DO 17 N=1,KC
      IF(ABS(TEMP).LT.EPSKIP)GO TO 17
      IF(ABS(AINV(I,N)).LT.EPSKIP)GO TO 30
      AINV(J,N)=AINV(J,N)-TEMP*AINV(I,N)
00030 CONTINUE
      IF(ABS(B(I,N)).LT.EPSKIP)GO TO 17
      B(J,N)=B(J,N)-TEMP*B(I,N)
00017 CONTINUE
00016 CONTINUE
00003 CONTINUE
      RETURN
00051 WRITE(5,52)
00052 FORMAT(5X,'THE MATRIX IS SINGULAR')
      IS=0
      RETURN
      END

      SUBROUTINE IDEMTX (A,N)
C      A IS N*N
      real A(N,N)
      DO  I = 1,N
        DO  J = 1,N
                   A(I,J) = 0.0
          IF(I.EQ.J) A(I,J) = 1.0
        end do
      end do
      RETURN
      END
```

Appendix E. *Developed* CIDL *Module*

```
let
  include array;
  include matrix;
  include matrixop;
  include noise;
#include "math.cdl";

-- VARIABLE LIST
  ddt                              : store(bool);  -- debug variable only
  iss                              : store(bool);
  five2rad                         : store(real);
  i1                               : store(real);  -- changed from integer
  k, iopt, ngr, j, c, npt, i2      : store(int);
  x, y, wx, wy                     = array_create(real, 100);
  xhatn, xhatold                   = matrix_create(real, 2, 1, 0.0);
  vx, vy                           = array_create(real, 100);
  z, xhat, zhat, nu                = matrix_create(real, 2, 1, 0.0);
  p, f, ft, r, pn, w, wt, pi       = matrix_create(real, 2, 2, 0.0);
  temp1, temp2, q, h, s, ht, si    = matrix_create(real, 2, 2, 0.0);
  xtilde                           = matrix_create(real, 2, 1, 0.0);
  xtildet                          = matrix_create(real, 1, 2, 0.0);
--  epsilon = matrix_create(real, 1, 1, 0.0);  -- there was a problem with dimension in matmul
  epsilon                          = matrix_create(real, 2, 2, 0.0);
  graph                            = matrix_create(int, 20, 200, 0.0);   --  was real
  nameg                            = array_create(string, 20);

-- OPEN FILES USED TO STORE DATA
  f8  : stream(char) = create(char, "kalman_plt");  -- need to use create instead of open
  f9  : stream(char) = create(char, "kalman_pl");
  f10 : stream(char) = create(char, "xtruth");
  f11 : stream(char) = create(char, "ytruth");
  f12 : stream(char) = create(char, "xmeas");
  f13 : stream(char) = create(char, "ymeas");
  f14 : stream(char) = create(char, "xest");
  f15 : stream(char) = create(char, "yest");
  f16 : stream(char) = create(char, "xnoise");
  f17 : stream(char) = create(char, "ynoise");
  f20 : stream(char) = create(char, "zxnoise");
  f21 : stream(char) = create(char, "zynoise");
  f22 : stream(char) = create(char, "p11");
  f23 : stream(char) = create(char, "p22");
  f24 : stream(char) = create(char, "epsilon");
  f25 : stream(char) = create(char, "xtilde");

 in
 -- put here to label the files
  format(f8,"KALMAN.PLT~%~%");
  format(f9,"KALMAN.PL~%~%");
  format(f10,"XTRUTH.PRN~%~%");
  format(f11,"YTRUTH.PRN~%~%");
  format(f12,"XMEAS.PRN~%~%");
  format(f13,"YMEAS.PRN~%~%");
  format(f14,"XEST.PRN~%~%");
  format(f15,"YEST.PRN~%~%");
  format(f16,"XNOISE.PRN~%~%");
  format(f17,"YNOISE.PRN~%~%");
  format(f20,"ZXNOISE.PRN~%~%");
  format(f21,"ZYNOISE.PRN~%~%");
  format(f22,"P11.PRN~%~%");
  format(f23,"P22.PRN~%~%");
  format(f24,"EPSILON.PRN~%~%");
  format(f25,"XTILDE.PRN~%~%");

  ddt := false; -- debugging flag

-- START MAIN PROGRAM
--i1 := 12345;  -- seed constant, too large, generates 0 < i < 12345
  i1 := 1.0;
  i2 := 54321;  -- seed constant
  c  := 73;     -- (NUMBER POINTS COLLECTED)

-- COMPUTE MODELING NOISE (SET TO ZERO)
  k  := 1;
  wx.initialize(0.0);  -- test only
  wy.initialize(0.0);  -- test only
  vx.initialize(0.0);  -- test only
  vy.initialize(0.0);  -- test only
  loop
    when content(k) <= content(c) =>
      wx.assign(k, makenoise(0, 0, 12, i1)); -- had to be rewritten completely
      wy.assign(k, makenoise(0, 0, 12, i1)); -- as an assign instead of passing
                                             -- the array element to change
      format(f16, "~d ~%", wx.index(k));
      format(f17, "~d ~%", wy.index(k));
      k := k + 1
  end loop;

-- SET INITIAL VALUES
  r.initialize(0.0);  -- test only
  x.assign(1, 10.0);
    if content(ddt) then             -- if statement for debugging/development
    format(terminal,"~%x array = ");
    x.print()
    end if;

  y.initialize(0.0);  -- test only
  y.assign(1, 0.0);
    if content(ddt) then             -- if statement for debugging/development
    format(terminal,"~%y array = ");
    y.print()
    end if;

  z.assign(1, 1, 10.0);
  z.assign(2, 1, 0.0);
    if content(ddt) then             -- if statement for debugging/development
```

```
     format(terminal,"~%z matrix = ");
     z.print()
     end if;

   p.assign(1, 1, 1.0);
   p.assign(1, 2, 0.0);
   p.assign(2, 1, 0.0);
   p.assign(2, 2, 1.0);
     if content(ddt) then              -- if statement for debugging/development
     format(terminal,"p matrix = ");
     p.print()
     end if;

five2rad := 2*math.pi*5.0/360;         -- the FORTRAN called for degrees, LISP uses radians
   f.assign(1, 1, math.cos(five2rad));
   f.assign(1, 2, -0.5 * math.sin(five2rad));
   f.assign(2, 1, 2 * math.sin(five2rad));
   f.assign(2, 2, math.cos(five2rad));
     if content(ddt) then              -- if statement for debugging/development
     format(terminal,"f matrix = ");
     f.print()
     end if;

   matrix_transpose(f,ft);
     if content(ddt) then              -- if statement for debugging/development
     format(terminal,"f matrix transpose= ");
     ft.print()
     end if;

--   q.initialize(0.0);  -- not needed
     if content(ddt) then              -- if statement for debugging/development
     format(terminal,"q matrix = ");
     q.print()
     end if;

   h.assign(1, 1, 1.0);
   h.assign(1, 2, 0.0);
   h.assign(2, 1, 0.0);
   h.assign(2, 2, 1.0);
     if content(ddt) then              -- if statement for debugging/development
     format(terminal,"h matrix = ");
     h.print()
     end if;

   matrix_transpose(h,ht);
     if content(ddt) then              -- if statement for debugging/development
     format(terminal,"h matrix transpose= ");
     ht.print()
     end if;

   r.assign(1, 2, 0.0);
   r.assign(2, 1, 0.0);
     if content(ddt) then              -- if statement for debugging/development
     format(terminal,"r matrix = ");
     r.print()
```

```
      end if;

    iss := false;

    xhatold.assign(1, 1, 10.0);
    xhatold.assign(2, 1, 0.0);
      if content(ddt) then            -- if statement for debugging/development
      format(terminal,"xhatold matrix = ");
      xhatold.print()
      end if;

-- START MAIN LOOP
    k := 1;
    loop

      -- WRITE INITIAL VALUES TO FILES
      when content(k) <= content(c) =>
        format(terminal, "~d~%", content(k)); -- monitor loop progress
        graph.assign(1, k, k);
        format(f10, "~d ~%", x.index(k));
        format(f11, "~d ~%", y.index(k));
        graph.assign(2, k, x.index(k));
        graph.assign(3, k, y.index(k));
        graph.assign(4, k, z.index(1, 1));
        graph.assign(5, k, z.index(2, 1));
        format(f12, "~d ~%", z.index(1, 1));
        format(f13, "~d ~%", z.index(2, 1));
        graph.assign(8, k, p.index(1, 1));
        graph.assign(9, k, p.index(2, 2));
        format(f22, "~d ~%", p.index(1, 1));
        format(f23, "~d ~%", p.index(2, 2));
        graph.assign(6, k, xhatold.index(1, 1));
        graph.assign(7, k, xhatold.index(2, 1));
        format(f14, "~d ~%", xhatold.index(1, 1));
        format(f15, "~d ~%", xhatold.index(2, 1));

    -- NORMALIZED STATE ERROR (EPSILON = XTILDET*PINV*XTILDE)
    -- XHATOLD = XHATN
        xtilde.assign(1, 1, x.index(k) - xhatold.index(1, 1));
        xtilde.assign(2, 1, y.index(k) - xhatold.index(2, 1));
        matrix_transpose(xtilde,xtildet);
              if content(ddt) and content(k) = 1 then -- if statement for debugging/development
              put("xtilde");
              xtilde.print();
              put("xtildet");
              xtildet.print()
              end if;
--      matrix_inv(p, pi, temp1); -- changed the temp1 to iss
        matrix_inv(p, pi, iss);
              if content(ddt) and content(k) = 1 then -- if statement for debugging/development
              put("p");
              p.print();
              put("pi");
              pi.print();
              put("iss");
```

```
          put(iss)
          end if;
     matrix_mult(xtildet, pi, temp1);
          if content(ddt) and content(k) = 1 then -- if statement for debugging/development
          put("xtildet");
          xtildet.print();
          put("pi");
          pi.print();
          put("temp1");
          temp1.print()
          end if;
     matrix_mult(temp1, xtilde, epsilon);
          if content(ddt) and content(k) = 1 then -- if statement for debugging/development
          put("temp1");
          temp1.print();
          put("xtilde");
          xtilde.print();
          put("epsilon");
          epsilon.print()
          end if;
     graph.assign(10, k, epsilon.index(1, 1));
     format(f24, "~d ~%", epsilon.index(1, 1));
     graph.assign(11, k, xtilde.index(1, 1));
     format(f25, "~d ~%", xtilde.index(1, 1));

-- STATE PREDICTION (XHAT = F*XHATOLD)
     xhat.assign(1, 1, f.index(1, 1)*xhatold.index(1, 1)
                       + f.index(1, 2)*xhatold.index(2, 1));
     xhat.assign(2, 1, f.index(2, 1)*xhatold.index(1, 1)
                       + f.index(2, 2)*xhatold.index(2, 1));
          if content(ddt) and content(k) = 1 then -- if statement for debugging/development
          put("xhat");
          xhat.print()
          end if;

-- TRUTH (X(K+1) = f*X(K))
     x.assign(k+1, f.index(1, 1)*x.index(k)
                 + f.index(1, 2)*y.index(k)
                 + wx.index(k));
     y.assign(k+1, f.index(2, 1)*x.index(k)
                 + f.index(2, 2)*y.index(k)
                 + wy.index(k));

-- COMPUTE MEASUREMENT NOISE
     r.assign(1, 1, 0.2*math.abs(x.index(k+1)));
     r.assign(2, 2, 0.2*math.abs(y.index(k+1)));  -- had r(r,1) not r(2,2)
     vx.assign(k + 1, makenoise(0, r.index(1, 1), 12, i1)); -- ';' error in type caused problem here
     vy.assign(k + 1, makenoise(0, r.index(2, 2), 12, i1));
     format(f20, "~d ~%", vx.index(k + 1));
     format(f21, "~d ~%", vy.index(k + 1));
     graph.assign(12, k+1, vx.index(k+1));
     graph.assign(13, k+1, vy.index(k+1));

-- MEASUREMENT (Z = X+V)
     z.assign(1, 1, x.index(k+1)+vx.index(k+1));
```

```
    z.assign(2, 1, y.index(k+1)+vy.index(k+1));

-- MEASUREMENT PREDICTION (ZHAT = H*XHAT)
    matrix_mult(h, xhat, zhat);

-- INNOVATION (NU = Z-ZHAT)
    matrix_sub(z, zhat, nu);

-- STATE PREDICTION COVARIANCE (P = F*P*FT+Q)
    matrix_mult(f, p, temp1);
    matrix_mult(temp1, ft, temp2);
    matrix_add(temp2, q, p);

-- INNOVATION COVARIANCE (S = H*P*HT+R)
    matrix_mult(h, p, temp1);
    matrix_mult(temp1, ht, temp2);
    matrix_add(temp2, r, s);

-- FILTER GAIN (W = P*HT*SI)
    -- matrix_inv(s, si, temp1);  --old
    matrix_inv(s, si, iss);
    matrix_mult(p, ht, temp1);
    matrix_mult(temp1, si, w);

-- UPDATE STATE COVARIANCE (PN = P-W*S*WT)
    matrix_transpose(w, wt);
    matrix_mult(w, s, temp1);
    matrix_mult(temp1, wt, temp2);
    matrix_sub(p, temp2, pn);
    p.assign(1, 1, pn.index(1, 1));
    p.assign(1, 2, pn.index(1, 2));
    p.assign(2, 1, pn.index(2, 1));
    p.assign(2, 2, pn.index(2, 2));

-- UPDATED STATE ESTIMATE (XHATN = XHAT+W*NU)
    matrix_mult(w, nu, temp1);
    matrix_add(xhat, temp1, xhatn);
    xhatold.assign(1, 1, xhatn.index(1, 1));
    xhatold.assign(2, 1, xhatn.index(2, 1));

  k := k + 1
end loop;

        if content(ddt) then             -- if statement for debugging/development
        put("graph");
        graph.print()  -- print loop result
        end if;

-- PLOT ROUTINE FOR OUTPUT DATA
    ngr := 13;
    nameg.assign(1, "k");
    nameg.assign(2, "x truth");
    nameg.assign(3, "y truth");
    nameg.assign(4, "x measured");
    nameg.assign(5, "y measured");
```

```
      nameg.assign(6, "x estimate");
      nameg.assign(7, "y estimate");
      nameg.assign(8, "p11, x covar");
      nameg.assign(9, "p22, y covar");
      nameg.assign(10, "epsilon");
      nameg.assign(11, "xtilde");
      nameg.assign(12, "meas noise x");
      nameg.assign(13, "meas noise y");

          if content(ddt) then            -- if statement for debugging/development
          put("nameg");
          nameg.print()
          end if;

      iopt := 1;
      npt  := 73;  -- related to c above
-- the cidl output statement is format()
-- looks like write() can also be used
      format(f8 ,"~d ~d ~d ~%", content(ngr), content(npt), content(iopt));
      -- write(8) (nameg.index(j), j := 1, ngr)           -- note FORTRAN use of assignment here
      j := 1;
      format(f8, "~a ~d ~%", nameg.index(j), content(ngr));
      j := 1;
      k := 1;
      format(f8, "~d ~d ~d ~%", graph.index(content(j), content(k)), content(npt), content(ngr));
      format(f9 ,"~d ~d ~d ~%", content(ngr), content(npt), content(iopt));
      j := 1;
      format(f9, "~a ~d ~%", nameg.index(j), content(ngr));
      j := 1;
      k := 1;
      format(f9, "~d ~d ~d ~%", graph.index(content(j), content(k)), content(npt), content(ngr));

  -- CLOSE DATA FILES
      close(f8);
      close(f9);
      close(f10);
      close(f11);
      close(f12);
      close(f13);
      close(f14);
      close(f15);
      close(f16);
      close(f17);
      -- close(f18);  -- this was not used
      -- close(f19);  -- this was not used
      close(f20);
      close(f21);
      close(f22);
      close(f23);
      close(f24);
      close(f25);
()
end let
```

Bibliography

1. "M. C. Escher: Twenty-Nine Master Prints,". Abrams, 100 Fifth Ave., New York, NY 10011, 1983. Photos by William Wegman.
2. Anderson, Christine and Merlin Dorfman, editors. *Aerospace Software Engineering, 136*. Progress in Astronautics and Aeronautics. 370 L'Enfant Promenade SW, Washington, D.C. 20024-2518: American Institute of Astronautics and Aeronautics, Inc., 1991.
3. Arnold, Robert S. "Software Reengineering," *IEEE Conference on Software Maintenance* (November 1990).
4. Bar-Shalom, Yaakov and Thomas E. Fortmann. *Tracking and Data Association, 179*. Mathematics in Science and Engineering. San Diego, California: Academic Press, Inc., 1988.
5. Biggerstaff, Ted J. "Design Recovery for Maintenance and Reuse," *IEEE Computer*, 36–49 (July 1989).
6. Brooks, Jr., Frederick P. *The Mythical Man-Month: Essays on Software Engineering*. New York, New York: McGraw-Hill Book Company, 1975.
7. Byrne, Eric J. *A Formal Process Model for Software Reengineering*. Contract Research, Kansas State University, November 1991.
8. Cardow, James E. and Eric J. Byrne. "Verification and Validation in the Re-engineering Process," *Publication Pending* (1992).
9. Chikofsky, Elliot J. and James H. Cross II. "Reverse Engineering and Design Recovery: A Taxonomy," *IEEE Software*, 13–17 (January 1990).
10. Chu, Wiliam and Sukesh Patel. "Software Restructuring by Enforcing Localization and Information Hiding," *IEEE Conference on Software Maintenance*, 42–49 (November 1992).
11. Hoare, C. A. R. *Communicating Sequential Processes*. Prentice-Hall, 1985.
12. Lewis, Ted. "Code Generators," *IEEE Software*, 67–69 (May 1990).
13. Lockheed Software Technology Center, Lockheed Palo Alto Research Laboratory, Organization 9610, Building 254E, 3251 Hanover Street, Palo Alto, CA 94304-1187. *Software User's Manual for the Automatic Programming Technologies for Avionics Software (APTAS) System*, June 1991.
14. Miller, Alan R. *Pascal Programs For Scientists and Engineers*. Berkeley, California: SYBEX, Inc., 1981.
15. Neighbors, James M. "The Draco Approach to Constructing Software from Reusable Components," *IEEE Transactions on Software Engineering, SE-10*(5):564–574 (September 1984).
16. Pressman, Roger S. *Software Engineering: A Practitioner's Approach*. New York, New York: McGraw-Hill Book Company, 1987.
17. Rugaber, Spencer, et al. "Recognizing Design Decisions in Programs," *IEEE Software*, 46–54 (January 1990).
18. Spivey, J. M. *Understanding Z*. Cambridge University Press, 1988.

Vita

Captain Chester A. Wright, Jr. was born on 21 September 1954 in Greenville, Mississippi. He graduated from Greenville High School in 1972, and in 1974 he joined the USAF. Upon completion of basic training, he received technical training at Chanute AFB, IL and was later stationed at Ellsworth AFB, SD. There he worked as a Missile Systems Analyst Specialist and a Maintenance Scheduler until November 1979. Captain Wright proceeded to Lowry AFB, CO and received technical training as a Precision Measuring Equipment Specialist and remained at Lowry as a Technical Instructor until January 1984. During this period he also received an Associate in Applied Science in General Electronics Technology from the Community College of the Air Force. He attended the University of Colorado at Denver and graduated with a Bachelor of Science in Electrical Engineering in August 1986. Upon graduating from Officer Training School in April 1987, he was assigned to the Electronic Security Command as an Operational Test and Evaluation manager. Captain Wright entered the School of Engineering, Air Force Institute of Technology, in May 1991.

Permanent address: 414 Coleman Street
Greenville, MS 38701

REPORT DOCUMENTATION PAGE	Form Approved OMB No 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE December 1992	3. REPORT TYPE AND DATES COVERED Master's Thesis

4. TITLE AND SUBTITLE Design Recovery for Software Library Population	5. FUNDING NUMBERS
6. AUTHOR(S) Chester A. Wright, Jr., Captain, USAF	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Air Force Institute of Technology, WPAFB OH 45433-6583	8. PERFORMING ORGANIZATION REPORT NUMBER AFIT/GCS/ENG/92D-23
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Wright Laboratories/AART Wright-Patterson AFB, OH 45433-6543	10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution unlimited	12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

The thesis research investigated design recovery as a means of populating a reuse library. The targeted library was part of the Automatic Programming Technologies for Avionics Systems (APTAS). APTAS uses a knowledge base of forms, to present questions to a user, and rules, to select the forms to present and choose existing library modules to use in composing a new system. The approach applied the reengineering model developed by Eric Byrne to accomplish planning for the project, expanded the renovation phase of this model to cover the actual design recovery, and applied the expanded model to populating the library.

Using the model in the project showed that design recovery is feasible in populating the library. However, if the recovered design could not be used directly, it could be used as a guide in developing new components. Additionally, certain modules make better candidates than others. Ideal candidates are self-contained in that they receive a value, perform a computation, and return a value. Once the module starts performing too many operations, expertise is required in the module behavior in order to separate the component for reuse.

14. SUBJECT TERMS Software Maintenance, Reverse Engineering, Design Recovery, CIDL, FORTRAN, APTAS	15. NUMBER OF PAGES 92
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL